



C 和 C++ 实务精选



C++ 面向对象多线程编程

Object-Oriented Multithreading Using C++

人民邮电出版社
POSTS & TELECOMMUNICATIONS PRESS

Cameron Hughes
Tracey Hughes
周良忠

著
译



C 和 C++ 实务精选

编写面向对象多线程应用程序的权威指南

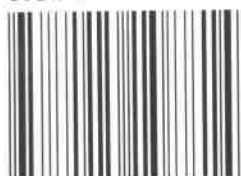
毫无疑问，这是我见过的最好的、最全面的多线程方面的图书。如果你把自己看作一位程序员，而不是组件装配员，那你就需要了解多线程方面的知识。只要不是刚刚入门的 C++ 程序员都可以从本书受益良多：不仅多线程编程的核心内容值得研习，书中的优秀源代码也可供借鉴。

—— Francis Glassborow, ACCU 主席

可通过 www.cloudcrown.com 下载的配套资源：

- 本书包含的所有源代码。
- 重要协议与信息资源。
- POSIX 线程管理规范的相关内容。

ISBN 7-115-10881-1



9 787115 108814 >

ISBN7-115-10881-1/TP·3200

定价：68.00 元

人民邮电出版社
<http://www.ptpress.com.cn>

C 和 C++ 实务精选

C++ 面向对象多线程编程

Cameron Hughes

Tracey Hughes 著

周良忠 译

人民邮电出版社

图书在版编目 (CIP) 数据

C++面向对象多线程编程/(美)休斯(Hughes,C), (美)休斯(Hughes,T.)著;
周良忠译. —北京:人民邮电出版社, 2003.4

ISBN 7-115-10881-1

I. C... II. ①休... ②休... ③周... III. C语言—程序设计 IV. TP312

中国版本图书馆CIP数据核字(2003)第018321号

版 权 声 明

Cameron Hughes, Tracey Hughes: Objected-Oriented Multithreading Using C++

Copyright© 1997 by Cameron Hughes and Tracey Hughes

All Rights Reserved.

Authorized translation from the English language edition published by John Wiley & Sons, Inc.

本书中文简体字版由 John Wiley & Sons 公司授权人民邮电出版社出版, 专有出版权属于人民邮电出版社。

版权所有, 侵权必究。

C 和 C++ 实务精选

C++ 面向对象多线程编程

◆ 著 Cameron Hughes Tracey Hughes
译 周良忠
责任编辑 陈冀康

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
读者热线 010-67132705
北京汉魂图文设计有限公司制作
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本: 800×1000 1/16
印张: 33.5
字数: 750千字 2003年4月第1版
印数: 1-4000册 2003年4月北京第1次印刷

著作权合同登记 图字: 01-2002-4865号

ISBN 7-115-10881-1/TP·3200

定价: 68.00元

本书如有印装质量问题, 请与本社联系 电话: (010) 67129223

内 容 提 要

全书共分 13 章，全面讲解构建多线程架构与增量多线程编程技术。第 1 章介绍了用于构建面向对象程序的不同类型 C++ 组件，以及如何使用这些组件来构建多线程架构。第 2、3、4 章简要介绍进程、线程、多任务处理、多线程化、规划以及线程优先权的概念。第 5 章讨论进程间和线程间通信。第 6 章讨论线程与进程同步与合作。第 6 章详细讨论临界区、死锁、数据竞争以及无限延迟方面的主题。第 7~10 章讲解用于线程同步、线程间通信、进程间通信以及多线程处理的 C++ 组件。第 11 章讨论 C++ 对象在多线程环境中的行为和交互方式。第 12 章简单介绍多线程应用程序的测试技术。第 13 章对全书内容进行扼要地回顾与思考。

本书适合用 C++ 创建多线程组件和应用框架的程序员阅读。

译者简介

周良忠，男，1970年生。本科毕业于武汉化工学院计算机应用专业。1995年毕业于中国科学院武汉岩土力学研究所，获硕士学位，1997年获得博士学位。1998年创办云巅工作室(<http://www.cloudcrown.com>)，为个人、中小企业提供专业软件定做服务。近几年开发了多款广受欢迎的共享软件。精通 C++、C#、Java、Perl 等开发语言。2001年开始从事计算机科技图书的创作和翻译工作，最新翻译力作有《C# Primer Plus 中文版》、《C++实践之路》等。



译者的话

据我所知，很多程序员惧怕多线程编程，主要因为多线程带来的死锁、数据竞争、优先权倒置、无限延迟等问题让他们望而却步。这些互斥性问题轻则让应用程序的速度不快反慢，重则使程序崩溃。而更多的开发人员义无反顾地拿起了“多线程”这把双刃剑。这也许是技术发展的大势所趋：

- 处理并行计算的需要；
- 随着计算机应用范围的拓展，需要超长运算时间的应用程序越来越多；
- 避免等待网络、文件系统、用户或其他 I/O 操作而耗费大量的执行时间。

如果能恰当运用多线程编程技术，以上领域的应用程序就会显著提高反应速度、处理的吞吐量以及处理器的有效利用率。当今的流行操作系统基本上都支持多线程。

但这方面能滋补读者的有价值的图书资料实在太少了。Cameron Hughes 与 Tracey Hughes 编撰的 *Object-Oriented Multithreading Using C++* 一书在国外读者中享有极高的声誉。能够翻译这样一本优秀的参考书实在是我的荣幸。在翻译过程中，我也为其中精彩的内容所吸引，甚至常常忘记了翻译的“本职工作”，径自捧读而爱不释手。

综观全书，它步步深入地剖析了面向对象多线程架构与增量多线程编程的概念与技术。通读全书，每一种多线程编程中可能遇到的互斥性问题都能找到令人满意的解决方案。因此，这本书将成为那些“惧怕”多线程编程程序员逾越技术障碍的有力武器。

本书与其他类似参考书相比，有两个突出不同点：一是本书所讨论的 C++ 组件和示例适用于 POSIX pthreads、Win32 API 或 OS/2 任何一种支持多线程的环境；二是丰富而又规范的图示能帮助读者更准确地理解所讲解的内容。另外，附录部分包括 POSIX 线程管理规范、同步类、线程类以及进程间通信类的源代码示例。www.cloudcrown.com 提供源代码下载。这些都是读者实践中的有用资源。

由于译者水平有限，错误在所难免，望广大读者不吝指正。译者 E-mail: web_zhou@21cn.com。

愿与广大读者共同学习、共同进步！

译者

2002 年 10 月

这是一本讲解用于生成可以同时执行多个任务的程序与应用的面向对象 C++ 组件构建的参考书。本书着重讲解支持多线程和并发处理的软件生成机制。本书推荐使用那些能自然适应于并行性的软件架构。本书所讲授的知识与软件“分治 (divide and conquer)”技术有关。

学习本书的必要性

当今的应用程序既需要大量的 CPU 时间，又依赖于高流量 I/O。用户不再满足于一次只执行一个任务的应用了。像 Web 浏览器这样支持上载与下载的应用程序必须允许用户在上载或下载的同时还能够撰写 E-mail 消息，或者阅览文档。电子数据表必须允许用户在浏览其中一部分占用大量处理器时间的图形的同时，还能执行另一部分冗长的计算。当今的许多应用程序都包含与声音同步的视频序列。视频序列的解码程序必须能够压缩或解压视频，同时要让视频图像与保存在视频图像中的声音数据同步。用户不能容忍无声的视频或只有声音没有图像的视频效果。假如字处理程序正在打印一份长文档的时候，用户不能编辑另外的文件，这是不可接受的。用户不会使用不能同时搜索和排序的数据库应用程序。如果某应用程序包含通信功能、打印功能、数据库功能、数值计算功能等等，用户希望能够并发执行其中的某些功能。一次只做一件事的日子已一去不复返了。我们需要那些可以让我们构建满足用户需求的应用程序的技术。我们需要那些可以让我们解决搜寻互联网上数据时所出现问题的技术。我们需要的是能应付处理器，而且能满足高流量 I/O 多媒体需求的技术和架构。

发展趋势

与之相应，应用程序的发展趋势就是让应用程序多线程化。多线程化应用程序是那些将任务分成多个线程的应用程序。这些线程就像工人一样。在单线程应用程序中，只有一个工人完成应

用程序将要满足的请求。如果用户发出多个请求，用户必须等待应用程序的唯一工人逐个地完成所有请求。不过，在多线程应用程序中存在多个工人。不仅有多名工人，而且这些工人还可以同时执行他们的功能。可分配工人组来共同完成一项任务，同时分配其他工人独立完成其他任务。在多线程应用程序中，将应用程序执行的任务分配给并发执行的工人集合，它们称做线程。如果用户发出了多个请求，这些请求可以同时得到满足，因为可以分派每个工人来满足用户请求的一部分。因为应用程序具有多个工人，所以工人可以并发满足多个请求。当应用程序包含多个线程时，该应用程序就是多线程处理的。

存在的难题

尽管多线程处理应用程序可以用来提高用户的生产力，也能加速程序的运行，但也存在许多缺陷。当应用程序拥有多个工人时，由谁来协调这些工人呢？如何将应用程序的功能性在这些工人间进行分派？哪一个工人在何时用多长时间来完成哪一项任务？谁首先启动？谁最后结束？如果有个工人的任务执行失败了，将会发生什么情况呢？如果工人混淆了信号与优先权，情况将如何？因为多线程程序可能包含上百，有时是上千个线程，所以我们可能应验这句谚语：人多误事 (Too many cooks spoil the stew)。也就是说，太多的工人可能导致应用程序执行迟缓，甚至被锁住。有时，工人间为谁首先使用特殊的资源而相互竞争。两个工人可能同时想发送不同的声音、图像或视频文件到同一个多媒体设备。一个工人可能想打开开关，而另一个工人却同时想关掉此开关。在一个应用程序中分派多个工人时可能遇到的难题有：

- 死锁；
- 数据竞争；
- 优先权倒置；
- 无限延迟。

这些难题是在构建多线程程序时必须避免的缺陷。它们带来了程序员经常要逾越的障碍。程序员通过一套构建完美的面向对象技术，可以获得设计和编写中到大型单线程应用程序的指导方针，但对于多线程应用程序，却没有这样的指导方针。软件开发者在构建多线程程序时通常要反复排除。构建包含多个线程的程序，没有好的蓝图必定导致灾难。甚至一个典型的中型单线程应用程序可能有上百个组件，每个组件由上千行代码组成。如果我们可以让所有组件协同工作而不发生故障，我们就可以判断该任务可以圆满完成。对于同样的中型应用程序，如果我们添加多个线程，我们通常会带来极大的复杂性。我们的目标可能只是加速应用程序，或者让它更符合用户的需要，但我们反而创建了一个脆弱的应用程序，容易遭受不可预测的崩溃，而且不可能对它进行维护。

解决方案

我们使用 C++ 组件来消除编写多线程程序时遇到的缺陷。我们通过面向对象技术来实现多线程应用程序。本书讨论用一种架构途径来构建多线程应用程序，还解释了如何构建用于实现支持

并发的软件机制的 C++ 组件。本书讨论的这类软件机制有：

- 互斥量对象；
- 事件互斥量对象；
- 匿名管道对象；
- 命名管道对象；
- 线程对象；
- 容器组件；
- 应用框架。

我们使用这些软件机制来探索增量多线程编程的概念，它使用封装、接口类、宿主类以及域类来构建多线程组件。然后组合多线程组件形成面向对象多线程架构。多线程架构成为构建可靠多线程应用程序的基础。为了使问题简单化，我们构建多线程应用程序时，一次只讨论一个组件。我们从构建支持并发的简单软件组件开始。将运行并发的组件与多线程化的宿主对象结合起来。再结合宿主对象与域对象形成多线程应用程序。

虽然本书主要是讲解 C++ 组件和面向对象技术，但必须注意，C++ 语言并不包含任何支持多线程编程的特定命令、语句或关键字。多线程能力由现在的操作系统提供。它假定 C++ 组件用于多线程环境中。现代操作环境，如 Solaris、Windows NT 以及 OS/2，它们都提供了支持多线程的系统服务。POSIX 标准 1003.1c 提供了针对线程的 API。针对线程的 POSIX 标准一般指 pthreads。C++ 语言可以利用这些环境所支持的任何线程系统服务。本书讨论的 C++ 组件可用于运行 pthreads 的任何环境、Win32 或 OS/2。

本书为谁而写

本书为软件工程师、软件开发人员、程序员、教育工作者、研究人员以及需要一种面向对象方法开发多线程程序和应用的學生而写。本书读者需要对 C++ 语言和标准 C++ 类库有一定的了解，而不必须具备多线程编程技术方面的经验。本书不是一本 C++ 编程或面向对象编程的指导书。它假定读者能够基本理解像封装、继承和多态这样的面向对象编程技术。本书介绍和解释不同类型的 C++ 组件。其中对操作系统进程的概念进行了介绍性地综述。我们还对核心线程和用户线程进行了详细的综述。定义和描述了多任务处理以及操作系统规划策略的概念。

本书的组织

第 1 章概览用于构建面向对象程序的不同类型 C++ 组件。第 1 章还简要介绍了如何使用这些组件来构建多线程架构。第 2、3 和 4 章对进程、线程、多任务处理、多线程化、规划以及线程优先权的概念进行了扼要的介绍。如果读者熟悉基本 C++ 类类型以及基本操作系统概念（这些都是理解线程的必需的知识），那么就可以跳过第 1 章到第 4 章的内容。第 5 章讨论进程间和线程通信。第 6 章讨论线程与进程同步以及合作。第 6 章应用了第 5 章讨论的概念，定义和讨论运行并发的程序中碰到的主要问题。也就是详细讨论了临界区、死锁、数据竞争以及无限延迟方面的主题。

第 7 章到第 10 章讲解用于线程同步、线程间通信、进程间通信以及多线程处理的 C++ 组件。第 11 章讨论 C++ 对象在多线程环境中的行为和交互方式。第 12 章给予读者测试多线程应用程序方面的提示。要应用我们所讨论的增量多线程处理技术，就必须完全理解第 5 章到第 11 章所讲解的知识。

类关系图

本书运用类关系图来描述相关类家族与集合、容器类层次之间的关系。对类关系图的详细解释参见配套资源中的附录 A。

支持的线程环境和编译器

本书中的所有主要代码示例都使用 POSIX pthreads、Win32 API 线程以及 OS/2 线程实现。尤其使用了针对 Linux 2.0 的 pthreads、Win32、OS/2 的核心线程。所有示例在运行 POSIX pthreads、Win32 API 或 OS/2 的任何环境中都能正常工作。虽然我们使用 Gnu C++、IBM 的 Visual Age 以及 Borland 的 C++ 来测试这些例子，但支持 ANSI/ISO C++ 标准的任何 C++ 编译器都可以拿来编译本书中的示例。而且，本书中的大部分示例还可以运行在 VMS 和 AIX 上。

测试与代码可靠性

虽然我们对本书中的所有示例和应用都进行了测试，以确保其正确性，但我们不能保证本书所包含的程序没有任何缺陷与错误、与任何特殊商业标准一致，或者满足任何特殊应用的要求。不要依赖于它们来解决问题，不正确的解决方案可能会伤害到应用者本人，或者导致利益上的损失。

读者由于使用本书中包含的示例或应用程序带来直接或间接的损失，作者与出版商并不承担任何赔偿义务。

致谢

感谢 Paul Mullins 博士、Ray Sweeney 教授、Raoul Hewitt 教授和 Debra Whitfield 博士，他们审阅了本书的初稿。还要感谢 IEEE 的 Mary Shepard，她帮助我们搜集 POSIX pthread 信息。特别要感谢的是我的家庭、朋友，以及在本书撰写过程中，参与“并行”深入讨论的相关人士。

目 录

第 1 章 C++组件简介	1
1.1 既是好消息，也是坏消息	1
1.2 面向对象方法	2
1.3 面向对象架构	2
1.4 C++组件	3
1.5 面向对象软件组件	3
1.5.1 什么是类	4
1.5.2 抽象数据类型	4
1.5.3 类作为模型	8
1.5.4 类类型	9
第 2 章 进程解剖	25
2.1 什么是进程	25
2.2 进程状态	28
2.3 进程优先权	34
2.4 上下文切换	35
2.5 进程关系	35
2.5.1 进程终止	39
2.5.2 同步和异步进程	39
2.6 进程映射	41
2.7 进程资源	42

2.7.1 硬件资源	43
2.7.2 数据资源	44
2.7.3 软件资源	44
2.7.4 优先权与资源	44
第3章 轻量级进程：线程	47
3.1 多线程处理	49
3.2 线程与进程的相似之处	52
3.3 线程与进程的不同之处	52
3.4 线程的优点	53
3.5 线程的缺点	54
3.6 线程类型	54
3.6.1 休眠 (sleeper) 和单步 (one-shot)	54
3.6.2 先占工作	55
3.6.3 延迟工作	55
3.7 线程相关信息	55
3.8 线程创建	57
3.8.1 谁可以终止线程	58
3.8.2 分离线程	60
3.8.3 远程线程	60
3.9 线程堆栈	60
3.10 线程控制	61
3.10.1 临界区	61
3.10.2 挂起和恢复线程	63
3.11 线程优先权	64
3.12 线程状态	70
3.13 线程与资源	71
3.14 线程的实现模型：用户级线程	71
3.14.1 核心级线程	71
3.14.2 混合线程	71
第4章 多任务与多线程编程	73
4.1 什么是多任务编程	73
4.1.1 对话级多任务编程	74
4.1.2 进程级多任务编程	74

4.1.3 多线程编程	75
4.2 合作和抢占式多任务	75
4.2.1 合作多任务	76
4.2.2 抢占式多任务	78
4.2.3 时间片的大小	79
4.3 多处理器下的多线程	80
4.3.1 非对称多处理器处理	81
4.3.2 对称多处理器处理	81
4.3.3 具有多处理器的多线程处理模型	82
4.4 规划策略	84
4.4.1 规划策略目标	84
4.4.2 规划策略准则	85
4.4.3 轮询和 FIFO 规划	86
4.4.4 最短任务优先规划法	87
4.4.5 最短剩余时间规划法	88
第 5 章 进程间和线程间通信	89
5.1 依赖关系	89
5.1.1 通信依赖性	90
5.1.2 合作依赖性	91
5.1.3 计数线程与进程依赖性	91
5.2 进程间和线程间通信	94
5.2.1 什么是进程间通信	94
5.2.2 进程间通信类型	95
5.3 线程间通信	114
第 6 章 合作与同步	123
6.1 竞争条件	123
6.1.1 数据同步	127
6.1.2 硬件同步	127
6.1.3 任务同步	128
6.2 同步关系	129
6.3 进程同步机制	130
6.3.1 信号量提供钥匙	130
6.3.2 信号量类型	132

6.3.3 自愿互斥量策略	136
6.3.4 使用互斥量锁定防止竞争条件	136
6.3.5 临界区	145
6.4 避免竞争条件	147
6.5 死锁必需的条件	148
6.6 远离死锁	148
第7章 接口类与进程间通信	149
7.1 接口类详解	149
7.1.1 接口类的类型	149
7.1.2 减小参数和全局变量的数量	155
7.2 C++没有多线程处理的关键字	157
7.3 面向对象接口到管道	158
7.4 使用接口类来实现面向对象命名管道	173
7.4.1 相关客户/服务器术语	174
7.4.2 名字包含哪些内容	174
7.4.3 命名管道和 <code>iostream</code> 复合	175
7.4.4 <code>npstream</code> 接口类	176
7.4.5 命名管道与 STL <code>istream_iterator</code> 和 <code>ostream_iterator</code>	181
第8章 同步对象	185
8.1 初识 <code>mutex</code> 类	186
8.1.1 命名互斥量类	195
8.1.2 同步和依赖性关系 (示例)	199
8.1.3 表示条件的类	206
8.1.4 等待多个事件或互斥量	213
8.1.5 通过类成员函数锁定和取消锁定	217
8.1.6 小结	219
第9章 线程处理面向对象架构	221
9.1 什么是多线程架构	221
9.2 使用多线程的常见架构	223
9.2.1 文件服务器	225
9.2.2 数据库服务器和事务服务器	227
9.2.3 应用服务器	229

9.2.4 事件驱动架构	233
9.3 黑板架构	235
9.4 途径上的不同(面向对象与过程化)	237
9.4.1 封装是关键(保护和数据隐藏)	239
9.4.2 类成员函数 CREW 策略	251
9.5 增量多线程处理	252
第 10 章 类层次和线程处理 C++ 组件	255
10.1 抽象基类	255
10.2 具体类——理想终结者	258
10.2.1 多线程层次中的节点类	260
10.2.2 线程与容器和集合类	261
10.2.3 应用框架类	276
第 11 章 类行为和线程处理	295
11.1 线程、对象和作用域	295
11.1.1 连接与作用域	296
11.1.2 线程和类作用域	297
11.2 同步关系和对象成员函数	297
11.3 在多线程环境中构建和析构对象	306
11.3.1 exit()和 abort()	306
11.3.2 构造函数和 SS 关系	307
11.3.3 析构函数与 FF 关系	308
11.3.4 线程集合与对象	309
11.3.5 线程与异常处理	311
11.4 线程安全函数	318
11.5 多线程环境中的不安全函数	319
11.6 在多线程架构中使用 STL 算法	320
第 12 章 测试多线程应用程序	323
12.1 软件测试的目标	323
12.1.1 分而治之(divide and conquer)	324
12.1.2 软件测试类型	326
12.1.3 对象的组件复合	327
12.1.4 成员函数访问数据组件	328

12.1.5 成员函数正确性	328
12.1.6 对象的过渡状态	329
12.1.7 成员函数调用序列	329
12.1.8 对象完整性	330
12.2 对象的测试实例	336
12.2.1 对象构建的测试实例	336
12.2.2 析构函数的测试实例	337
12.2.3 赋值的测试实例	338
12.2.4 对象派生子类	339
12.2.5 成员函数性能的测试实例	339
12.2.6 对象资源需求和测试实例	340
12.2.7 测试公有对象访问、受保护对象访问以及线程化对象访问	340
12.3 测试多线程架构的问题	341
12.3.1 开放层次问题	341
12.3.2 规划问题	341
12.4 使用常用模型和架构	342
第 13 章 实现并发的最后思考	345
附录 A POSIX 线程管理规范	349
附录 B 类关系图规范	387
附录 C POSIX 线程管理函数	391
附录 D Win32 线程管理函数	415
附录 E OS/2 线程管理函数	439
附录 F 线程和同步类 (POSIX, Win32 以及 OS/2)	473
参考文献	503
索引	509

C++组件简介

相对于普通语言的一般表达能力而言，通过这种表示法可以让我们更清楚地表述我们希望表达的内容。同时可以准确表达复杂的思想而避免歧义，这是日常语言不能轻松解决的。不过，为以上优点必须付出代价：必须了解不熟悉的特性；必须掌握一种新的表示法。

Symbolic Notation, Haddock's Eyes and the Dog-Walking Ordinance
——Ernest Nagel¹

最近 POSIX 线程标准在许多 UNIX 环境中（例如 Sunsoft 的 Solaris、IBM 的 AIX 和 Linux）的实现，以及 OS/2、NT 等其他多线程操作系统的成功，它们都为 C++程序员提供了真正的多线程环境。多任务处理（multitasking）允许同一时刻执行多个进程，而多线程处理（multithreading）允许一个进程在同一时刻执行多个任务。单个进程中执行的每个任务称做一个线程（thread）。当一个进程使用了多个线程时，该进程就称做多线程化（multithreaded）。程序员通过多线程可以将一个进程分解成一系列可以同时执行的线程。如果进程运行于拥有多个处理器（processor）的计算机上，每个线程可以在单独的处理器上执行。借此，程序员可以实现一个程序中能够并行执行的任务。

1.1 既是好消息，也是坏消息

多线程环境的实现带给 C++程序员的既是好消息，也是坏消息。好消息是，由于为 C++程序员提供了软件设计和软件开发阶段应用并发（concurrency）和并行（parallelism）操作的能力，多线程为解决软件操作性能问题和软件组织问题开辟了一条全新的途径。对于 C++程序员来说，并

¹译者注：摘自 Ernest Nagel 发表于 *World of Mathematics* 杂志上的文章 *Symbolic Notation, Haddock's Eyes and the Dog-Walking Ordinance*。

行除了提供面向对象模型外，还提供了另一种强大的模型。

坏消息是，并发编程和并行编程可能难度相当大。多线程环境引入了单线程环境不具有的许多缺陷。因为程序指令在单线程环境中一次只执行一条，所以程序员不必担心存在两条竞争、可能相互排斥的程序指令。任何需要访问数据或某些设备的程序指令必须排队等待访问数据或设备的机会。另一方面，多线程环境可能使多程序指令试图在同一时刻访问同一块数据或资源。这种情况可能导致死锁（deadlock）、无限延迟（indefinite postponement）以及数据竞争（data race）。当两个或更多的线程紧密关联，而且无法进一步执行时，就会发生死锁。死锁将导致应用程序或系统的停顿。无限延迟指线程 A 一直等待线程 B 的完成，然后才能继续自己的处理。但糟糕的是，线程 B 永远不能完成它该做的事，这时就产生了无限延迟。所以，线程 A 将一直等待永远也不会发生事件的执行。不幸的是，为了解决这一问题，通常需要强行删除所有线程，使得进程将非正常终止。在并发编程时，当多个线程试图同时更新同一个数据块时，就会发生最后一个恼人的问题：数据竞争。谁应该首先访问目标数据？应首先使用哪一部分更新？多线程编程带来表现力和操作性能好处的同时，死锁、无限延迟以及数据竞争这些不利因素总是威胁着某些应用程序，使它们出现意外不正常终止。

1.2 面向对象方法

同样，面向对象编程技术用于解决许多过程编程技术（procedural programming technique）难以解决的问题（Meyer, 1988），面向对象架构（object-oriented architecture）可用于管理并行编程情况下出现的死锁、无限延迟以及数据竞争问题。本书通过使用面向对象组件和面向对象架构介绍了增量多线程编程（incremental multithreading）的概念。使用面向对象组件来模块化应用程序可能需要的线程需求。使用封装（encapsulation）和对对象访问策略（object access policy）来简化并发和并行编程。我们建议在面向对象应用程序中使用多线程的最有效途径是通过构建于 C++ 进程间通信组件（IPC）、C++ 同步组件以及 C++ 互斥组件之上的应用框架（application framework）。我们建议构建 IPC、同步和互斥组件的最有效途径是构建接口类（interface class）和类库（class library），让它们封装提供线程和 API 任务的操作系统服务。通过使用接口类、类库以及多线程应用框架，C++ 程序员可以构建充分利用多线程、多处理器环境的面向对象架构。

1.3 面向对象架构

软件片断的架构是一套主宰软件操作的规则（rule）、模式（pattern）、进程（process）以及断言（assertion）。软件架构表示与数据组织和执行流程有关的整体结构。架构反映了设计思想（philosophy）、开发方法学（methodology）以及域模型（domain model）。当软件片断基础结构（infrastructure）基于代码/数据封装、代码/数据继承和多态时，此软件就称做面向对象架构（object-oriented architecture）。具有面向对象架构的软件构建于类（class）和类层次（class hierarchy）。面向对象架构已经成功运用于控制各种规模的软件设计和开发中的复杂性。面向对象架构使用组件方法来构建软件。每个组件都是一个独立的软件部分，它具备特定的功能。将这些软件部分装

配起来形成一个更大的软件部分，或生成一个完整的应用程序。面向对象架构被证明是最有效的软件片断打包和组织方式之一。在 C++ 语言中，面向对象架构构建于 C++ 组件。

1.4 C++组件

C++ 语言中对面向对象编程的支持使得它成为最富生命力的面向对象环境之一。因为针对 C++ 语言制定了 ANSI/ISO 标准，所以，面向对象编程更具有规范性。尽管 C++ 可以作为 C 的改良语种进行使用，但它的真正优势在于支持封装（encapsulation）、继承（inheritance）、多态（polymorphism）和参数化类型（parameterized type）。C++ 程序员通过这些特征可以开发面向对象软件组件。

1.5 面向对象软件组件

面向对象软件组件是独立的功能性单元。每一个组件均是某些人物、地点、事件或思想的软件模型。每个组件包含一些数据和服务列表，或包含可以操作以上数据的运算。组件可以为外界提供部分、或所有的数据与服务，也可能完全不提供。能够被外界使用的数据和服务称做组件的接口（interface）。程序员或其他软件片断访问组件的唯一途径是通过组件的接口。虽然组件是独立的，但它们并不是单独完备的程序或应用。组件是用于构建程序、应用，甚至其他组件的基石。例如，对于一个典型的专家系统，它至少包含 5 个部分：

- 推论引擎；
- 知识库；
- 知识获取组件；
- 解释组件；
- 用户接口。

每个部分都表示专家系统内的一个独立模块。当专家系统用 Lisp 或 C 之类的语言编写时，这些模块将作为报告给其他函数的函数集来实现。这种语言使用自顶向下（top-down）技术来实现专家系统，将导致程序分解难以实现，专家系统难以测试和维护。相反，如果专家系统的这 5 个基本部分用面向对象架构来完成的话，5 个模块就可以用一个或多个在对象消息模型上下文内通信的 C++ 组件来实现。这样，C++ 组件就可以模拟专家系统内的自然结构和关系了。虽然每个 C++ 组件是独立的，但它们不能单独作为一个完整的程序或应用，认识这一点很重要。通过某种配置，这些组件实现一个专家系统。不过，如果应用在其他组合体中，它同样容易形成另一个完全不同类型的应用，如一个互联网 Web 浏览器。C++ 软件组件是一种软拼装体（soft-legoware），它用于构建应用于多目的、多种类应用以及可能无限制组合体的一般性软件部分。常用的 C++ 组件有 4 种：

- 类（class）；
- 集合和容器（collection and container）；
- 类库（class library）；

- 应用框架 (application framework)。

这些 C++ 组件为大型和小型面向对象应用程序和系统开发提供了主要的基石。这些组件赋予了应用程序真正的面向对象架构。使用这些基石，我们可以引入面向对象架构、增量多线程以及多线程架构的思想。我们将在第 9 章对增量多线程和多线程架构进行详细的讨论。在第 10 章，我们将详细讨论 C++ 组件，在第 9 章详细讨论面向对象架构。图 1-1 显示了赋予应用程序面向对象架构的 C++ 组件层。

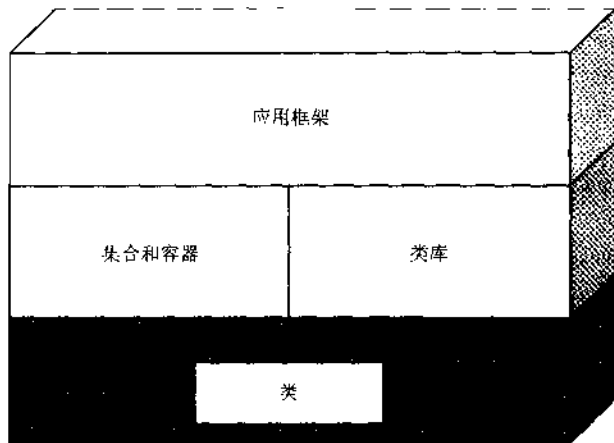


图 1-1 赋予应用程序面向对象架构的 C++ 组件层

虽然这些层并非每个面向对象架构都具备，但它们通常存在于大型 C++ 开发项目中。随着更多基本 C++ 组件的使用，应用程序的面向对象架构逐渐发展。请注意图 1-1 中 C++ 组件的关系。应用框架使用集合类 (collection class)、容器类 (container class)、类库 (class library) 以及基本类 (basic class)。集合、容器和类库组成了更基本的类。类组件是所有其他 C++ 组件的基础。

1.5.1 什么是类

类 (class) 是定义人物、地点、事物或思想的一套特征和行为。类按组放置对象或概念。组的每个成员共享一套公共属性。这些属性定义了该组。例如，如果我们要讨论类“三角形”，可以使用“三条边”、“三角之和为 180°”这样的属性来定义这个类。如果 X 表示所有可能的特征和行为的集合，那么类就是包含特定属性和行为、而排除了其他属性和行为的 X 子集。正是由于排除了其他一些属性且包含了另外一些属性，才使得这个类具有唯一性。这些属性用于区分各个类。这些类可以是非常抽象，也可以是非常具体。它们可以是人为虚构的，也可以是现实生活中自然存在的类。类可用于实现 ADT (abstract data type, 抽象数据类型)。类也可以用于为现实对象建模。

1.5.2 抽象数据类型

类可用于实现抽象数据类型。数据类型是数据值以及操纵该数据的一套运算的集合。抽象数据类型定义数据类型的基本概念。抽象数据类型是定义数据类型的数学概念 (Tenebaum, Langsam,

Augenstein, 1990)。抽象数据类型与特定硬件或软件实现问题无关。抽象数据类型的目的是指定与数据概念以及该数据概念可用服务有关的结构 (structure)、规则 (rule) 和断言 (assertion)。抽象数据类型描述了与特定编程语言无关的特定数据类型的基本概念。无论是内置数据类型还是用户自定义类型都具有该级别的抽象。抽象数据类型描述了属性以及由结构提供的外界可用服务。操作或服务以及数据值集被捆绑在一起创建一个单元，这就是类。在 C++ 中，用户自定义类型使用 `class`、`struct` 或 `union` 来创建。本书的示例使用类 (class) 结构来实现用户自定义数据类型。类具体表达了数据类型的布局。类定义可能作用于数据值集的所有行为和 (或) 操作的一切代码。

当在 C++ 中用类实现 ADT 时，类扩展了该语言的内置类型。例如，C++ 语言的基本数据类型中没有有理数 (rational number)，但是，使用 C++ 的类概念，程序员可以实现有理数。恰当运用运算符重载 (operator overloading)、构造函数 (constructor) 以及析构函数 (destructor)，程序员实现的新有理数类可以像 `int`、`float` 和 `double` 内置数值类型一样使用。也就是说，程序员可以执行有理数算术运算。程序员可以声明有理数数组、将它们作为参数传递等等。例如：

```
rational X(3,4);
rational Y(6,5);
rational Z[2];
Z[0].assign(1,2);
Z[1].assign(3,9);
cout<<X+Y+Z[1];
```

以上代码声明了两个有理数 X、Y，以及一个有理数组 Z。然后将有理数相加，并将 37/12 的结果发送到标准输出。

可以设计和实现程序员定义的 ADT，让程序员的数据类型比内置数据类型更完善。例如，C++ 整数运算允许程序员相除任意两个整数。C++ 的 `int` 定义没有阻止程序员用 0 来除。尽管程序员可以根据需要自由使用 `int`，但对于整数数据类型是不允许被 0 除的，因为被 0 除超出了整数数据类型的语义。

不仅 ADT 的实现提供了 ADT 可用的服务，而且这一实现可以加强对 ADT 的服务语义。例如，不会定义 `int` 类型被 0 除。使用 `int` 类型除法运算的先决条件和断言强制限制被 0 除。程序清单 1-1 中的 `rational` 类定义了针对有理数对象的服务。

程序清单 1-1 用户自定义 `rational` 类的声明

```
1  #include <iostream.h>
2  #include "eclasses.h"
3
4
5  class rational{
6  protected:
7      long Numerator;
8      long Denominator;
9      general_exception Exception;
10 public:
11      rational(long Num = 0, long Den = 1);
```

```

12     rational(const rational &X);
13     virtual void assign(long X, long Y);
14     rational operator*(const rational &X);
15     rational operator+(const rational &X);
16     rational &operator=(const rational &X);
17     rational operator-(const rational &X);
18     rational operator/(const rational &X);
19     int operator==(rational X);
20     long numerator(void) const;
21     long denominator(void) const;
22     friend ostream &operator<<(ostream &out, rational X);
23     void reduce(void);
24 };

```

对象的构造函数不允许 0 作为分母；更新有理对象分母值的 `assign()` 成员函数不接受 0 值。`rational` 类重载的/运算符不允许被 0 除。程序清单 1-2 演示了 `rational` 类中的成员函数如何强制实现 `rational` 数据类型的语义。

程序清单 1-2 `rational` 类的一些定义，演示了如何强制实现 `rational` 类的语义

```

1  rational::rational(long Num,long Den)
2  {
3      Numerator = Num;
4      if(Den == 0){
5          Exception.message("Zero Is A Invalid Denominator");
6          throw Exception;
7      }
8      Denominator = Den;
9  }
10
11  rational rational::operator/(const rational &X)
12  {
13      if(X.Denominator == 0){
14          Exception.message("Divison By Zero Undefined");
15          throw Exception;
16      }
17      rational Temp;
18      Temp.Numerator = Numerator * X.Denominator;
19      Temp.Denominator = Denominator * X.Numerator;
20      Temp.reduce();
21      return Temp;
22  }
23
24  void rational::assign(long X, long Y)
25  {
26
27      if(Y == 0){

```



```

28     Exception.message("Zero Is A Invalid Denominator");
29     throw Exception;
30 }
31 Numerator = X;
32 Denominator = Y;
33 }
34

```

程序员不能试图用 0 除,或被其他成员函数给分母赋值 0 让系统崩溃。一旦引入了 0 这个值,就抛出异常。相对于一些内置数值数据类型,这类强制性赋予了用户自定义数据类型更好的功能性。实现 ADT 来扩展 C++ 提供的基本数据类型是使用类结构的主要原因之一。

虽然 C++ 允许程序员实现传统的 ADT,但类结构也允许程序员实现更特殊的 ADT。例如,在多媒体环境中,就有机会使用程序员自定义的数据类型。从文本到语音的转换引入了语音 font 数据类型,其中每种 font 代表计算机用来从语音上输出文本的一种不同声音。有男声语音数据类型和女声数据类型。对这些用户自定义数据类型的操作包括更改 font 的语调、font 的语音,以及在不同标点类型间的停顿。在多媒体编程环境中,声音、动画以及视频处理与传统编程环境中的数值和字符串处理一样是必需的。就像传统编程环境中需要用户自定义数据类型一样,多媒体环境也需要实现 ADT,如调色板类型、波形类型、用户自定义 font、位图类型以及电影类。多媒体数据类型扩展了传统 ADT 概念。C++ 组件可用于实现这一新层次的 ADT。程序清单 1-3 中位图类(bitmap class)表示一个位图数据类型。

程序清单 1-3 用户自定义位图数据类型示例

```

class bitmap{
protected:
    HWND Handle;
    HBITMAP BmHandle;
    HPS PSpace;
    char *Graphic;
    BitMapFileHeader Header;
    BitMapInfoHeader2 Info1;
    BitMapInfo2 Info2;
public:
    bitmap(HWND Han);
    ~bitmap(void);
    virtual void readBitmap(void);
    virtual void displayBitmap(void);
};

```

对这一个类的操作包括使用重载--运算符倒置位图,以及使用重载++运算符拉伸位图。因为希望 bitmap 类能像内置数据类型一样使用,所以为此类定义了赋值运算符。程序员可以声明数组,或堆栈,或位图类型的对象集。位图是多媒体应用程序中的重要部分,而且在传统的数据库设置中也可以找到。通过将 ADT 位图实现为 C++ 组件,程序员就可以在几乎所有需要操纵和显示位图数据的面向对象环境中自由插入这个组件。作为 ADT 实现的 C++ 类仅是类结构的一个重要用途。

利用 C++ 对面向对象编程支持的另一个威力显示在使用类作为模型的实现中。

1.5.3 类作为模型

类可用于创建软件模型。软件模型有两种主要类型，C++ 可以用于创建这两种模型。第一类模型是一些过程、概念或思想的放大表示。这种模型用于分析或试验。例如，C++ 可用于分子建模。在这类建模中，对分子内结构和化学过程的假设可以在使用了支持面向对象 C++ 的软件中实现。可以模拟化学键间的距离和角度。当引入一组新原子时，可以研究此时分子的行为。可以评价化学反应如何影响分子结构。这些假设可以不借助真实的分子而在计算中探索。软件模型模拟了真实分子的形成和作用，所以可以用于预测真实分子结构和化学反应的特征和行为。

第二类软件模型是用软件对某些现实任务、过程或思想的再现。这一模型的目的是让它像真实对应物、系统或应用程序的一部分一样发挥作用。软件取代人工系统或某些物理体的某些组件。例如，我们可以使用 C++ 的面向对象能力为桌面或袖珍计算器建模。我们可以使用类结构来声明软件计算器的组件。软件计算器于是就可以将桌面计算器取而代之了。这类模型与用于试验和分析的模型相反。软件计算器实际上取代了桌面计算器。类 `calculator` 复制了桌面计算器的所有功能性特征，这是为了取代它，而不是为了研究它。

注意，实现抽象数据类型的类与作为模型的类之间的区别。一般而言，数据类型用于支持编程工作。即程序员可以使用 `float`、`int`、`char`、`bool` 或 `rational` 等，让它们各施其责。就像钉住面板的钉子一样。这些钉子用于帮助建造房子，但它们不是房子结构的一部分。当 C++ 类用作模型时，类通常是编程过程的最终结果，或者表示最终结果架构的重要部分。用作模型的类不仅仅是一种数据类型。模型化类实际上作为现实人物、地点、事物或思想的替身。软件模型抓住了真实事物的本质。图 1-2 描述了一个简单的袖珍计算器以及计算器所具备的组件的基本逻辑模型。

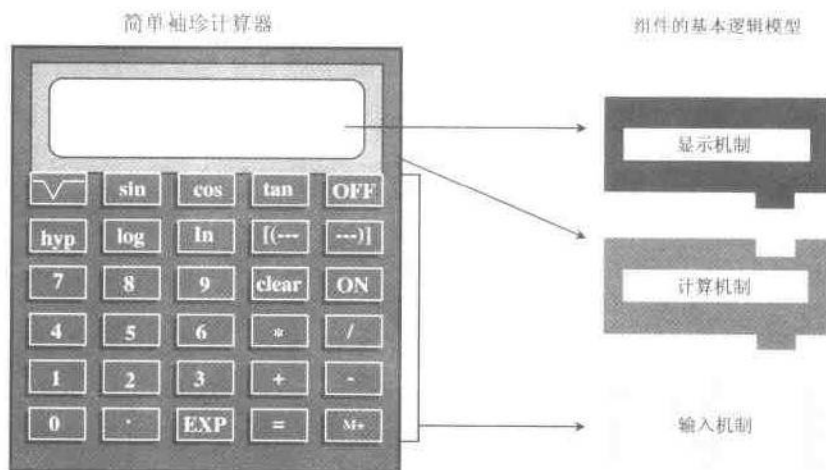


图 1-2 简单袖珍计算器映射到计算器组件的逻辑模型

请注意，计算器可以分解成 3 个基本逻辑部分。每个袖珍计算器都有显示机制、输入机制以

及一些执行计算和算术运算的机制。如果我们可以构建 C++ 组件来为这些基本逻辑组件来建模，那我们将得到一个软件计算器。它带给我们有关建模的重要一点。我们不仅必须决定正在建模的事物或过程的基本逻辑组件是什么，而且必须对正在建模复制其功能性的组件有充分的理解。这需要我们进一步研究计算器基本逻辑组件。图 1-3 显示了对基本袖珍计算器逻辑组件的更基本分解。

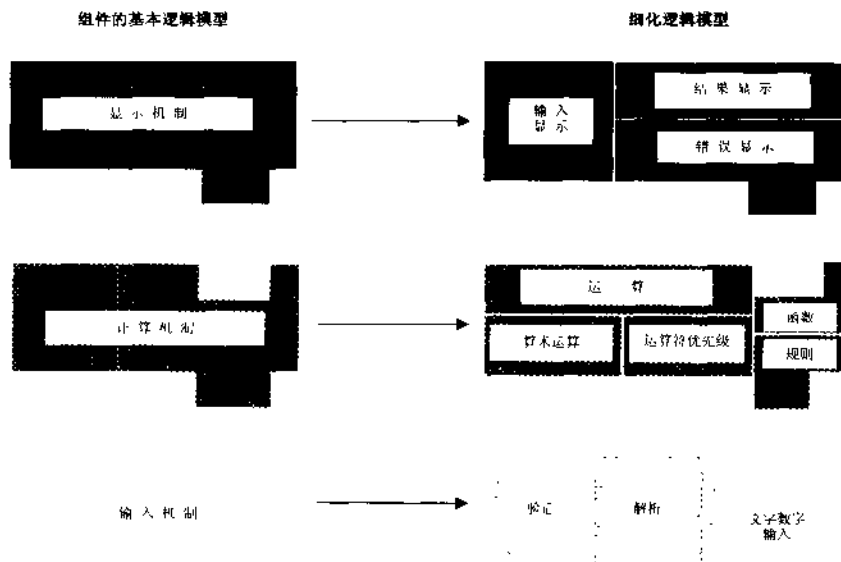


图 1-3 基本袖珍计算器的逻辑组件的主要分解

输入组件被分解成包括解析、验证及数值和文字数字检查几个部分。显示机制必须负责输入的显示，以及显示输出和任何可能产生的错误消息。计算机机制显示了对算术规则、运算符优先级、算术运算符以及函数调用的需要。如果我们使用一个 C++ 类来完成这些组件，实际上就是抓住了简单袖珍计算器的本质。

显示机制、输入机制以及计算机机制的特征都可以使用 C++ 对面向对象编程的支持来获得。图 1-4 显示了模型中逻辑组件与用于实现逻辑模型的 C++ 组件声明间的关系。

图 1-4 中类 `calculator` 是 C++ 组件思想的一个快照。它包含若干个 C++ 组件，而且它本身就是一个 C++ 组件。它具有真正的面向对象架构。C++ 类是所有更复杂 C++ 组件的基石。虽然只有一个类结构，但有多种可以在 C++ 中实现的逻辑类。

1.5.4 类类型

C++ 中的“类”这个词有两种用法。第一种用法指由关键字 `class` 表示的 C++ 语言结构。这个词的第二种用法指封装起来形成对象的一套行为和属性的逻辑思想。类在开发面向对象应用程序或类库中作为许多不同的功能。我们可以基于类在面向对象开发中的不同使用方式来讨论不同的类类型。一些类类型仅对用作蓝图类（`blueprint class`）有用，这些蓝图类为其他类提供推荐接口策略。某些类仅能用作祖先类（`ancestor class`）或基类（`base class`）。一些类类型不能成为好的基

类，所以不应当用于继承层次中。常用类类型有 8 种：

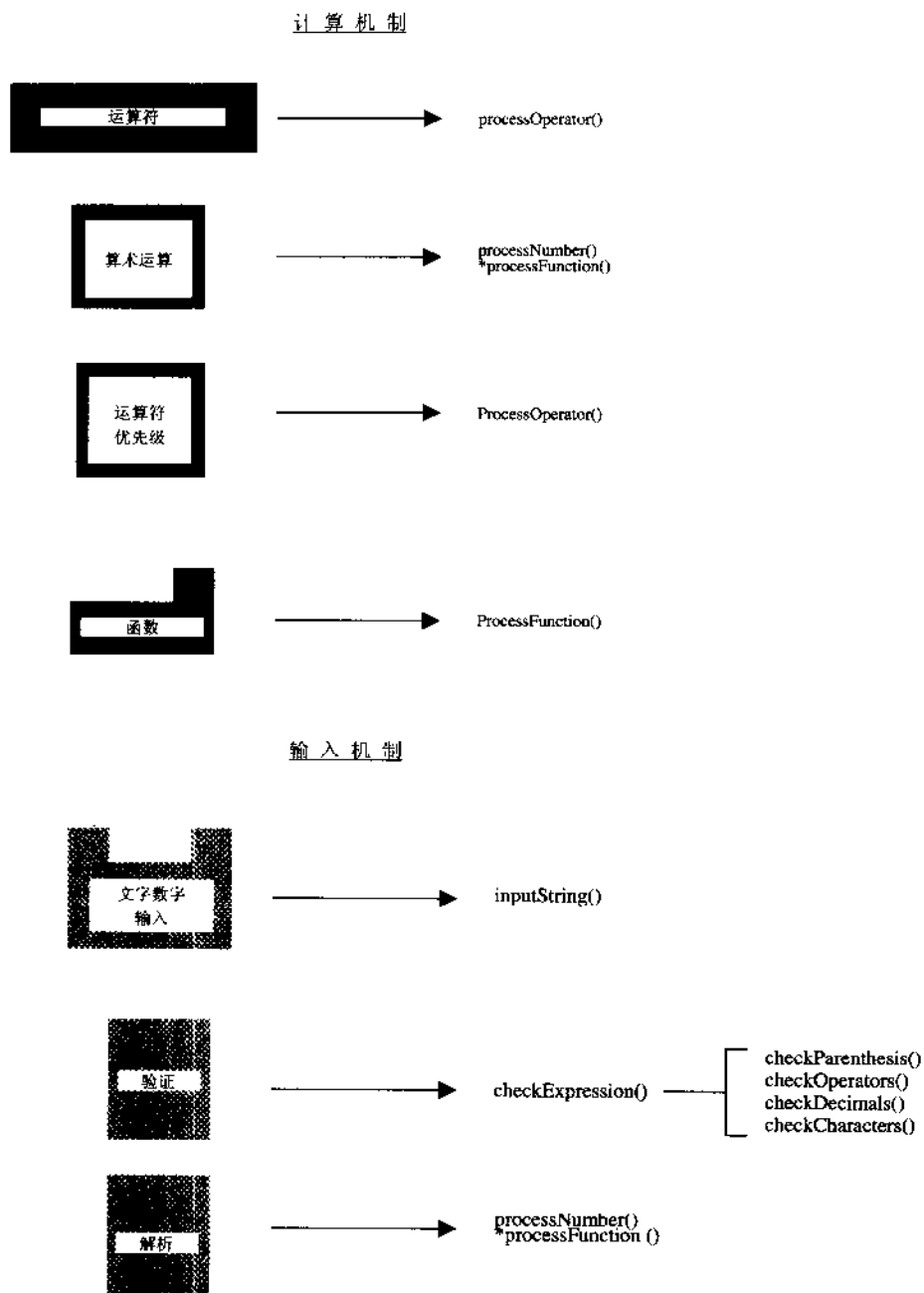


图 1-4 模型中的运算、输入机制与用于实现它们的 C++组件间的关系

- 具体类 (concrete class);
- 抽象类 (abstract class);
- 接口类 (interface class);
- 节点类 (node class);
- 支持类 (support class);
- 域类 (domain class);
- 应用类 (utility class);
- 集合和容器类 (collection and container class)。

大部分面向对象开发工作都应用了以上部分或全部基本类别的类。请记住，这些类并不表示特定的语言结构，而是用于实现逻辑类的技术。表 1-1 描述了 8 种常见的类类型，每种类类型在面向对象架构中都有独特的功能。

表 1-1 8 种常见的类类型

类类型 (class type)	描 述
具体类	独立类；表示一个祖先-后代世系终止的结束类
抽象类	为所有子类提供布局和蓝图的基类。不能声明此类型的对象；必须派生新类，对抽象基类中声明任何虚函数提供定义
接口类	用于修改或增强另一个类或类集合的接口。为了为非面向对象代码和数据提供面向对象性而用来封装独立函数
节点类	提供了继承和多态的基础；不包含纯虚函数
域类	创建类在指定域内模拟部分现实或实体
支持/应用类	不管在任何域内，对于不同的应用都非常有用
集合和容器类	其他对象组的一般性容器

一、具体类

虽然 C++ 中的许多类作为可供使用的基类或祖类而设计，但具体类是作为结束类而设计和实现的。一般而言，具体类为独立类，它表示祖先-后代世系的终止。“典型情况下，具体类并不适合于相关类一般系统。每个具体类不必参考其他类就可以单独理解。”(Stroustrup, 1991) 虽然具体类可能构建于其他类，但具体类是进一步继承的终点。在大部分情况下，将具体类作为基类或祖类是没有意义的。程序清单 1-1 中的 `rational` 类就是具体类的一个好例子。这个类没有设计为一般性类，而只是针对有理数而已。难以将这个类用作基类。注意，这个类具有受保护数据成员和虚成员函数 `assign()`。受保护数据成员和虚成员函数意味着将来可能的继承。在 `rational` 类声明中包含它们表示一种“安全总比出纰漏”的策略。自然而然，总会存在这样的可能：某些应用程序需要 `rational` 类的一些奇异子类，但这反映的只是例外，而不是规则。一旦实现具体类后，它们就不会在后代类中特殊化。具体类可以在它的实现中得到优化，因为没有必要担心派生类中的功能性 (Stroustrup, 1991, 431-435)。

从第 10 章可见, 具体类是线程处理中多线程类的理想候选者。因为具体类定义为结束类, 所以它线程化时, 不必考虑派生类中出现的覆盖或重载重要线程化成员函数。具体类可以用于支持第 10 章讨论所讨论的增量多线程编程的概念。具体类通常比其他类类型更容易使用, 因为具体类不需要用户在使用它前实现成员函数。一些类, 如容器类, 则需要用户实现比较运算符、赋值运算符等等。具体类的完整性可以拿来就用, 就效率而言, 它的操作性能与内置数据类型等同。具体类试图将最小化对其他类的依赖性。也就是说, 具体类可以有基类, 仅仅是在需要时如此。

二、抽象类

抽象类是为它的所有后代提供蓝图的类。事实上, 抽象类只是一个蓝图。用户不能声明一个抽象类的对象。为了使用抽象类, 用户必须首先从基类派生一个新类, 而且为抽象类中声明的所有纯虚函数提供实际定义。缺省情况下, 抽象类必须至少有一个后代, 它才会发挥作用。这与具体类截然不同。具体类不支持子类, 但使用抽象类时要求有子类。具体类类型不是一种特殊语言结构。它是一种设计逻辑类的技术和策略。不过, 抽象类也不是一种语言结构, 但它被 C++ 语言结构所支持。为了让一个类成为抽象类, 至少必须有一个纯虚函数。纯虚函数形式如下:

```
virtual return type function() =0;
```

例如, 类 A 有两个纯虚成员函数 lock()、unlock() 和一个虚析构函数:

```
class A{
public:
    virtual void lock(void)=0;
    virtual void unlock(void)=0;
    virtual ~A(void);
}
```

将函数 lock() 和 unlock() 初始化为 0 使它们成为纯虚函数。没有 0 这个初始化器, 它们仅仅是虚函数。类 A 实际上没有定义这些函数; 它仅声明了它们。因为具有纯虚函数任何类不能用于创建运行时对象, 为了使用这个类, 用户必须从这个抽象类派生一个类, 而且为抽象基类声明的每个虚函数提供函数定义。例如, B 从 A 派生, 并提供了两个纯虚函数的定义:

```
class B: public A{
protected:
    pthread_mutex_t X;
public:
    B(void);
    ~B(void);
    virtual void lock(void) {pthread_mutex_lock(x);}
    virtual void unlock(void) {pthread_mutex_unlock(x);}
}
```

不能进行如下声明:

```
A MemoryLock; //非法
```

因为 A 为抽象类。不能从抽象类创建运行时对象。可以进行下面的声明:

```
B MemoryLock; //合法
```

因为类 B 通过调用 POSIX 函数 pthread_mutex_lock() 和 pthread_mutex_unlock() 已经实现了两个纯虚函数 lock() 和 unlock()。如果类 B 只实现了其中一个纯虚函数, 那么, 类 B 也将被认为是

一个抽象类，因为它仍然包含一个纯虚函数。纯虚函数充当实现策略，它的任何用户在使用前必须实现纯虚函数。从概念上讲，纯虚函数的语义将在子类的实现中得到保持。也就是说，用户所定义的函数 `lock()` 和 `unlock()` 对“锁定”有合理的解释。用户不应该让定义的 `lock()` 和 `unlock()` 毫无意义。例如，若将 `lock()` 和 `unlock()` 定义为加和减函数就违背了锁定和取消锁概念的语义。

抽象类对于提供模式、蓝图和后代类遵循的原则有用。如果遵循了蓝图的语义，后代类的行为可能按抽象类提供者和使用者的期望那样。通过使用抽象类，C++程序员可以提供 C++ 组件的规范，在它的构建中指导组件的实现者。

三、接口类

类用户可用的数据成员和成员函数组成了该类的接口。C++对于类有 3 种访问策略。数据成员和成员函数可以是：

- 私有 (`private`)；
- 受保护 (`protected`)；
- 公有 (`public`)。

1. 私有策略

当数据成员和成员函数为私有时，它们仅能被中间类的成员函数所访问。这意味着私有数据成员和成员函数不能被非该类的成员函数所访问。这种限制扩展到这个类的后代上。例如，在下面代码中：

```
class A{
private:
    semaphore X;
};

class B: public A{
};
```

类 B 继承类 A。但是，类 B 不能访问类 A 的 X 数据成员。这个数据成员被声明为私有，它不是类 A 接口的一部分。私有访问策略表明，私有数据成员及成员函数不能被继承，所以不是类的接口的一部分。

2. 唯一成员访问 (`members-only access`)

类的数据成员和成员函数可以声明为受保护。当类的成员受保护时，它们可以被类的中间成员及类的任何后代所访问。与私有策略相反，声明为受保护的成员可以被任何子类访问。仅能被中间类和中间类的后代访问的数据成员和成员函数称做类的受保护接口 (`protected interface`)。虽然类的后代可以访问，但类层次之外的函数不能访问。例如，在下面的代码中：

```
class A{
protected:
    semaphore X;
public:
    A(void);
}
```

```
main{
    A Temp;
    Temp.X.lock(); //对受保护数据成员的非法调用
}
```

Temp 在类型 A 的对象中声明。A 将 X 作为一个受保护数据成员。X 有一个 lock() 成员函数。虽然 Temp 是类型 A 的一个对象，但函数 main() 不能访问 X 数据成员或它的成员函数，因为 X 为受保护，访问权只限制于家族函数。

3. 对公共开放

类可以将成员函数和数据成员声明为公有 (public)。声明为公有的数据成员和成员函数可以被任何人访问，不管它们是不是类的成员。这些数据成员和成员函数组成了类的公有接口。

四、什么是接口类

接口类用于修改和增强另一个类或类集合的接口。修改后，类可能更易于使用、功能更强、更安全或语义上更正确。接口类调整和完善接口，使它更有用或更有效。接口调整的例子有：更改函数或数据成员名；更改数据类型、返回类型或参数列表等等。接口类的一个好的例子是容器适配器 (container adaptor)，它是标准模板库 (Standard Template Library, STL) 的一部分。

适配器为来自标准模板库的列表、矢量和双端队列容器提供了公有接口。例如，程序清单 1-4 中的 stack 类用作一个修改 vector 类的接口类。

程序清单 1-4 STL stack 类的声明

```
1 template <class Container>
2 class stack {
3     friend bool operator==(const stack<Container>& x, const stack<Container>& y);
4     friend bool operator<(const stack<Container>& x, const stack<Container>& y);
5     public:
6         typedef Container::value_type value_type;
7         typedef Container::size_type size_type;
8     protected:
9         Container c;
10    public:
11        bool empty() const { return c.empty(); }
12        size_type size() const { return c.size(); }
13        value_type& top() { return c.back(); }
14        const value_type& top() const { return c.back(); }
15        void push(const value_type& x) { c.push_back(x); }
16        void pop() { c.pop_back(); }
17    };
```

通过声明：

```
stack<vector<T>>>Stack;
```

stack 类为用户提供了—个语义正确的类。堆栈的正式概念指弹出和推入堆栈的操作。堆栈是一种 LIFO (后进先出) 数据结构，意思是，最后推入堆栈的数据块将是首先从堆栈中被弹出的数

据块。top 操作返回弹出堆栈的下一个项。请注意程序清单 1-4 中的 13 行、14 行、15 行和 16 行，top()、push() 和 pop() 操作只是调用其他成员函数，如 c.pop_back()、c.push_back() 以及 c.back()。stack 类通过调整类 c（在这里就是 vector 容器）提供的名字，让用户能使用更熟悉的操作名字。最终通过对 vector 公有接口的修改使得应用了堆栈适配器的代码更容易理解、维护或调试。

从程序清单 1-4 可见，接口类有时依靠使用内联函数（inline function）。接口类继承或包含该类，然后定义将要作出的调整。以程序清单 1-4 中第 9 行为例，c 是一个被 stack 类解释了接口并重命名了的类。

接口类也可以充当非任何类成员函数的包装器。当我们包装在线程处理、进程间通信和多任务处理中使用的操作系统基本要素时，接口类就会发挥作用。通过封装操作系统基本要素，为用户提供这些基本要素的一般性接口，使代码跨平台的移植性更强。例如，我们可以声明一个类 mutex，它为用户提供与平台无关的锁定和取消锁服务：

POSIX

```
class mutex{
protected:
    pthread_mutex_lock X;
public:
    lock(){pthread_mutex_lock(&X);}
    unlock(){pthread_mutex_unlock(&X);}
};
```

OS/2

```
class mutex{
protected:
    HMTX X;
public:
    lock(){DosRequestMutexSem(X,1000);}
    unlock(){DosReleaseMutexSem(X);}
};
```

mutex 类是一个接口类。它为 UNIX 和 OS/2 环境下的操作系统服务提供了新接口 lock() 和 unlock()。在这里，接口至少有两种重要功能：第一个功能是为跨平台 mutex 类用户提供一个一致性的接口；第二个功能是对用户隐藏操作系统细节。使用 mutex 类时，程序员不必熟悉任何操作指定调用所需的语法或参数。

我们将在本书中经常使用接口类来封装特定操作系统调用和参数，为敏感或重要部分提供保护，而且为线程处理、进程间和多任务处理功能提供公共接口。

五、节点类

也许在 C++ 编程中使用的最强大类就是节点类，它是继承和多态的基础。与抽象基类一样，节点类设计用于继承。与抽象基类不一样的是，节点类不包含纯抽象虚函数。节点类可以即时使用。不过，节点类的设计是着眼于将来的。它被设计成可重用类。它提供了可以在派生类中覆盖的虚成员函数。

节点类提供了可以被派生类继承的受保护数据成员和成员函数。它使用基类的指针允许在后

代类中操纵成员函数。用户可以通过多态和继承特殊化节点类。节点类既可以是基类，也可以是派生类。节点类为类层次提供了实质内容。

六、域类

域类是一种创建用来模拟指定域内实体的类。类的意义特定于该域。域类为现实的某些方面建模。它捕获某些现实过程或概念的规则、断言和行为。域类为特定应用或特定系统的类。例如，在一个税务应用中，必须构建一个与此域相关的类，如 1040 纳税申报表类与 W-2 申报表类。又如在太空飞船模拟器中，域类包括 rocket 类、fuel 类、weather 类以及 pilot 类。

域类与编程支持类、用户接口类或数据库类完全分离，除非该域为系统编程。域类可以抽象基类、具体类或节点类。面向对象应用程序有一个表示应用程序核心的域类层次。例如，在创建一个用于计算税费的应用程序中，1040_tax_form 类可以继承一个名叫 tax_form 的基类，还包含一个名叫 deductible 的类。

七、支持/应用类 (Support/Utility Class)

支持/应用类与域无关，可以跨域使用。应用类在不同的应用程序中非常有用。例如，一个应用类可以为日期类、时间/时钟类，或判断操作或任务执行时长的计量类。这类功能在软件安装、文件下载或上载等时发挥作用。另一类支持/应用类是用户接口类。例如，一个列表框类或一个滚动条类。

八、集合和容器类

集合和容器为充当其他对象组一般性容纳器的对象。集合和容器类可能只是类库的一部分，或者一个类库可能只包含集合和容器类。虽然应用框架通常应用集合和容器，但集合和容器并不是应用框架。集合和容器为一般目的的分组结构。开发者或设计者可以使用集合和容器来操纵不同种类或相同种类的对象组。集合和容器用于管理对象组的方式与传统数组用于管理传统数据类型（如整数或字符）的方式是一样的。

集合和容器为对象。这意味着继承、多态和封装的优点都可以应用于集合和容器。其中的许多结构是面向对象版本：

Stack	List
Queue	Associative Array
Deque	Graph
Set	Tree
Multiset	Table

不过，集合和容器类设计的可能性已超越了这些传统数据结构。集合和容器类可以用于实现域类。我们碰到的有：

Garage	Vehicle
Bank	Ring
Cell	Paragraph
Crowd	Cabinet
Transfinite Set	Warehouse
Room	Tube

Group Field

实现集合和容器类的两个途径

实现集合和容器类有两个重要途径：

- 基于对象途径；
- 基于模板途径。

虽然两种途径的最终结果都是支持容器功能、插入操作、删除操作和完全对象访问的集合和容器类，但两种途径提供了根本不同的设计和开发思想。每种途径都有自己的内存管理策略、代码管理策略、效率平衡机制以及类组件结构性。基于对象结构非常依赖于多态和继承，产生了高度纵向的集合和容器结构；而基于模板的结构十分依赖于 C++ 中模板结构的一般性功能，并产生集合和容器的横向结构。模板途径使用了所谓的参数化编程（parameterized programming）。基于对象途径则倾向于需要类关系和类层次的深层知识，并且需要程序员通过虚成员函数特殊化集合或容器。基于模板途径侧重于要求程序员理解 FAT 接口（Stroustrup, 1991, 452），而且要求程序员为集合和容器开发通用的接口。

基于对象途径从一般性开始，然后通过继承推进到具体化。基于模板途径通过 FAT 接口从具体化推进到一般性。所使用的构建技术通常是设计思想和效率间的一种折衷。使用一定面向对象思想的规范，基于对象结构是最适当的途径。对于使用类来为现实概念建模和模拟的项目尤其如此。对于这些项目，使类接口尽可能逼真地为现实进行建模和模拟是关键所在。纵向类层次可以包含大量信息。基于对象层次充当分类学图谱，在集合的层次化结构中，它用作描述域和概念知识的信息表示机制（Hughes, 1996）。对于其他条件和需求，基于模板结构则是最适合的。没有一种单一、最优的途径来构建集合或容器类。具体实现由开发它的所在环境以及使用目标环境所决定。

基于对象途径的一个典型例子是 NIH（National Institute of Health，（美国）国家卫生研究所）集合和容器类（Gorlen, Sanford, Plexico, 1991）。在 NIH 类库中，任何指定某集合或容器的对象都必须直接或间接为类 Object 的后代。同样，任何容器类都必须直接或间接派生于类 collection。NIH 类库的基于对象类层次如图 1-5 所示。

一般性类 Object 的简化版在 C++ 中通过关键字 class 来定义。

```
class Object{
protected:
    string ClassDescription;
public:
    Object(void);
    string description(void);
    void description(string S);
};
```

一旦我们有了一般性 Object 类，我们就可以声明需要放置在容器中作为类型 Object 后代的任何类。例如，如果我们打算在一个 vector 类中保存一个 rational 类型的对象时，则必须让 rational 继承于 Object：

```
class rational: public Object{
};
```

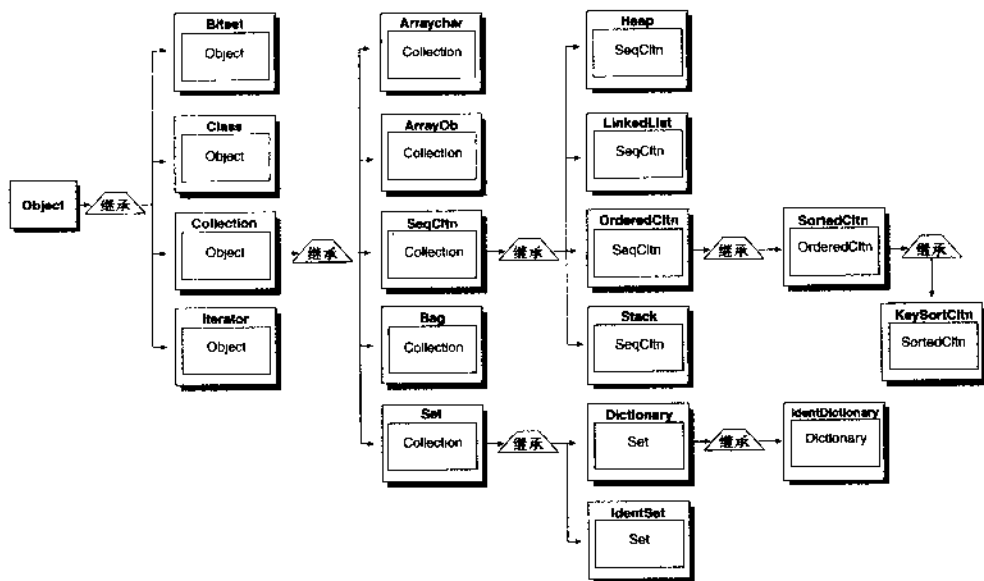



图 1-5 NIH 类库的基于对象类层次

类集合通常提供应用于所有容器类的虚成员函数（Gorlen, 152）。例如，集合类将提供虚成员函数来：

- 在容器中插入或添加对象；
- 从容器中删除对象；
- 返回容器中的对象数；
- 返回容器的容量；
- 测试空容器；
- 测试容器中的特定对象。

我们可以声明集合类的一个简化版本，如下所示：

```

class collection{
protected:
    int Size;
public:
    collection(int Sz){Size=Sz};
    virtual void insert(Object & X)=0;
    virtual int size(void)=0;
    virtual int empty(void)=0;
};
  
```

集合类的这一简化版本中包含有纯虚函数。这意味着我们不能声明对象为 collection 类型，因为任何具有纯虚函数的类称做抽象基类。抽象基类不能有直接实例（instance）。不过，抽象基类充当所有后代类的最小化实现策略。继承了抽象基类，且因为基类中的纯虚性而不能定义成员函数的任何类本身就是一个抽象基类。一旦派生类拥有了所有纯虚性的定义，那这个派生类就有了

一个运行时实例。

假如我们希望得到一个矢量容器，它可以表示容纳 `rational` 类，声明如下：

```
//基于对象 Vector 示例
class vector: public collection:
protected:
    Object *V;
    int Pos;
public:
    vector(int Sz);
    ~vector(void);
    void insert(Object *X);
    int size(void);
    int empty(void);
    Object operator[](int N);
};
```

然后我们可以声明 `rational` 类为 `Object` 类型的后代，`vector` 为 `Collection` 类型。这样我们可以在矢量中插入有理数（通过 `insert()`），因为矢量为集合的后代，集合类知道如何插入从类型 `Object` 中继承的后代。

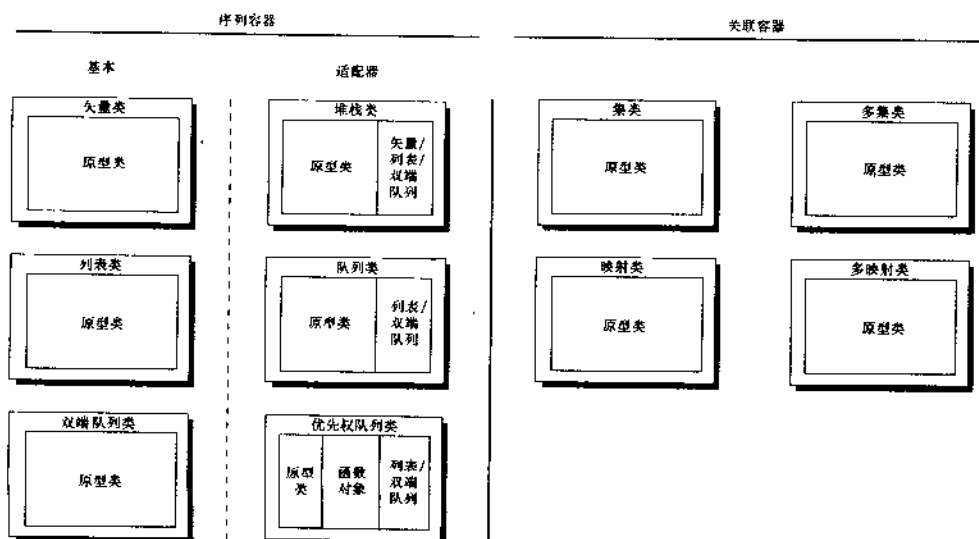
模板途径的一个典型例子是标准模板库（STL）的实现，STL 是 C++ 语言的一部分。它们没有形成一个纵向层次，相反 STL 类具有横向关系。图 1-6 描述了 STL 的结构。

STL 中的容器都是参数化的。例如，使用 STL 声明一个有理数的矢量，声明如下所示：

```
vector<rational> Myvector
```

STL 现在不是多线程化的，所以在多线程架构中使用 STL 时要小心谨慎。另一方面，基于对象途径却为线程化架构的思想提供了一些支持。第 9 章将对多线程架构进行详细讨论。

a.



b.

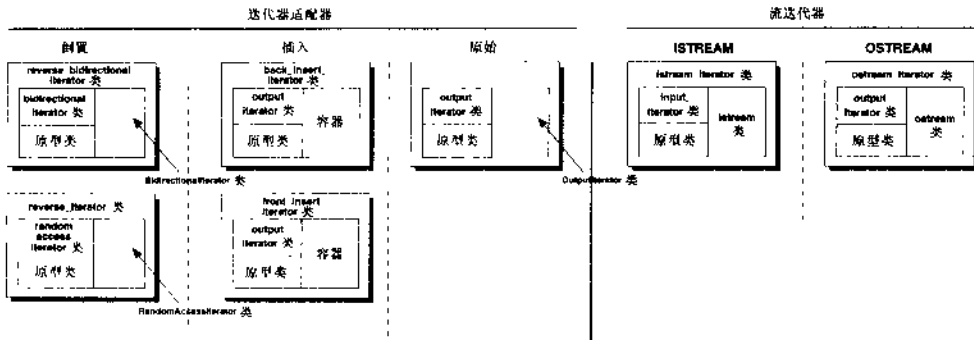


图 1-6 STL 的平面结构

九、类库

类库在代码重用规划中也很重要。类库是通常按类别分组的有用集合和一般性类。就像函数和过程库一样，C++类库也有许多类别：

- 数学类库；
- 用户接口类库；
- 文件管理类库；
- 内存管理类库；
- 图形类库；
- 应用类库；
- 任务和协同程序类库；
- 操作系统 API 类库；
- 特定域类库。

类库（class library）和应用框架（application framework）之间是有区别的。类库可用于它可以应用的几乎任何方面，但应用框架是一个类的集合，它有预定义的结构，并表示一个应用。应用框架中类之间的关系是事先制定的，应用框架的使用也只限于它所表示应用的范围之内。类库中的类根本没有任何关联，只是不同的类别，而且它们的使用模式不可预测。例如，在数学类库中，也许有三角形（triangle）类和多项式（polynomial）类。它们之间没有预定义关系，但可以创建一种关系。而且，多项式类或三角形类可以某种合适的方式使用于任何应用中。对于应用框架却并不如此。虽然应用框架代表着广泛的应用领域，但应用的组件有预定义关系，而且应该基于一套预设计的假设和规则协同工作。类库为类的集合，同时，应用框架也是类的集合。与应用框架的使用方式相比，类库的使用方式更宽广。

与 C++ 支持的其他面向对象组件一样，类库也可以通过继承来扩展。这一特征赋予开发者重用的强大模型。开发者可以按原样选择类库，实现代码重用的优势，也可以通过派生新类并特殊化这些新类的行为来扩展类库的功能。由于类库支持对象陷喻，所以开发者没有必要关心类的内部实现。开发者不必在类库中修改源代码来扩展它的功能。

C++环境中可用的最重要类库之一是 iostream 类库。iostream 类库包含帮助程序员完成输入/输出服务的不同 C++ 组件。iostream 类库为程序员提供了一种 I/O 的面向对象途径。iostream 提供了缓冲器 (buffer)、流 (stream)、格式化 (formatting) 以及 I/O 块的模型。iostream 类库由 3 种基本类别组成:

- 流状态类;
- 缓冲器类;
- 转换/翻译类。

图 1-7 中的类关系树显示了 iostream 的类层次。

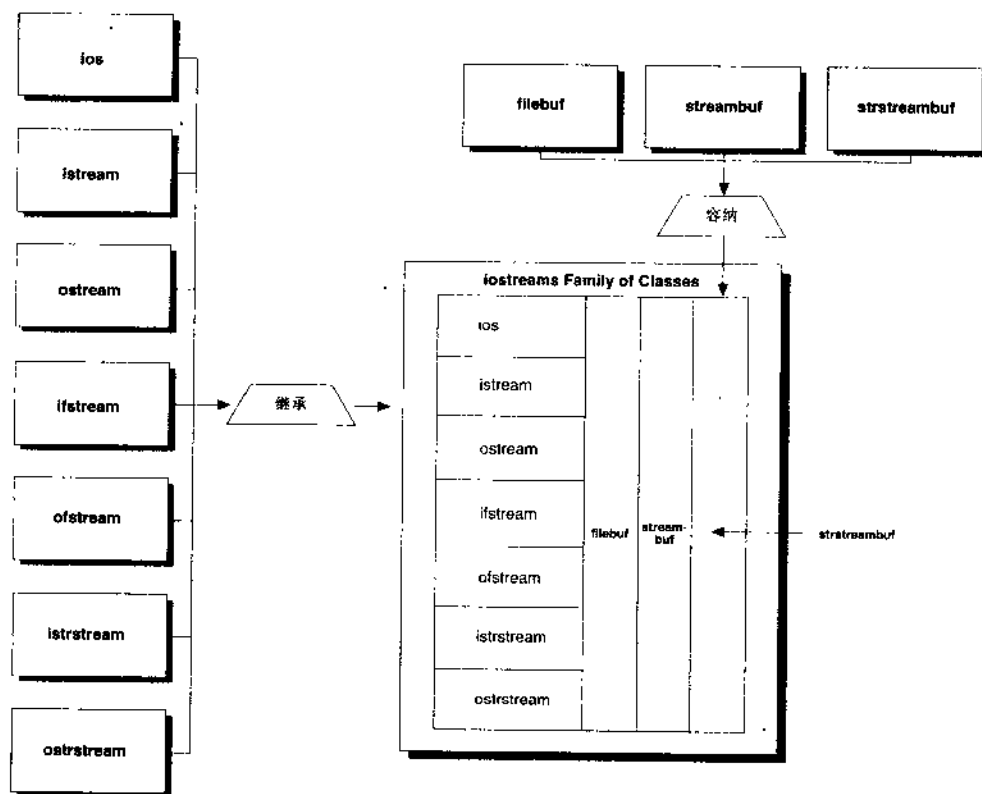


图 1-7 iostream 类家族的类关系层次

iostream 中的组件可以由程序员处理几乎任何所需 I/O 服务来扩展。iostream 类库是一种典型的设计优良、可扩展性强的 C++ 类库。它是一种优秀的类库，是因为它封装了许多有用函数，这些函数可以在广泛的应用中执行，因此节省了程序员的时间和精力。3 种基本领域——流状态、缓冲器和数据类型转换——都可以由 iostream 类库层次来处理。流状态类由 ios 组件来表示。缓冲器类由 streambuf 类家族来表示。转换/翻译类由 istream 和 ostream 类家族表示。ios 组件为面向对象流建立模型。streambuf 组件充当插入或提取 I/O 中对象的临时放置区域。istream 和 ostream

类家族执行从对象到字节流、从字节流到对象的相互转换。

事实证明，`iostream` 类库中的若干组件在构建多线程、进程间通信以及线程间通信基石时是有用的。在第 7 章，我们将使用 `filebuf` 组件和 `istream`、`ostream` 组件来帮助给面向对象管道（object-oriented pipe）建模，同时封装线程间和进程间通信。

十、应用框架

计算机程序通常是特定问题的一般解决之道。例如，以典型的工资单程序为例，它计算一些公司中职员净小时工资。程序的计算步骤如下：

```
(While) 当还有职员未计算时
    取得下一个职员
    计算总固定工时
    计算总加班工时
    RegPay = 总固定工时 * 每小时工资
    OvtPay = (每小时工资 * 1.5) * 总加班工时
    TotPay = RegPay + OvtPay
    NetPay = TotPay - 折扣
End While
```

这个程序解决的特定问题是计算每个职员的净工资额。该程序通过指定一般性的过程，它不仅计算一个职员，而且计算所有职员的净工资额，因此提供了一种一般性解决方案。这是通过使用固定工时、加班工时、固定工资、加班工资、净工资、总工资、折扣变量（这些变量可以代表所有职员的数据），同时揭示这些变量对所有小时职员都成立的关系来实现的。通过提供对特定问题的一般性解决方案，该程序就可以针对不同的职员群、不同的工作作息体系、不同的工资水平、也可能是不同公司的职员来重复使用。这个程序的力量在于，不同量之间的关系是正确的，即使这些量的值发生变化时也是如此。换句话说，输入可能改变，但输入间的关系保持不变，所以，我们针对特定问题得到了一般性解决方案。

按对特定问题提供一般性解决方案的同样方式，应用框架为问题类提供了特定解决方案。也就是说，一个应用框架代表了整个范围内的程序，这些程序都按相似的方式解决或表示问题。换个方式讲，一个应用框架代表一系列问题的单一解决方案。应用框架是一般性面向对象应用程序。应用框架作为整个应用程序的一种模式。它具体化应用程序即将具有的基本结构或骨架，但没有提供应用细节。应用框架指定了面向对象架构中软件部分间的关系、工作模式以及协议。面向对象语言处理器就是应用框架的一个好例子。它所解决的特定问题是获取输入语言并将此语言翻译成某些输出形式。这一框架包含的软件部分有：

- 确认组件；
- 符号化组件；
- 解析器组件；
- 语法分析组件；
- 词汇分析组件；
- 规则。

这些软件部分可以组合形成一种非常熟悉的工作模式：

```

void main(void){
    language process X;
    X.getString()
    X.validateString(0
    X.parseString()
    X.lexicalGroup(0
    ...
    ...
    ...
    X.result()
}

```

这一特定的解决方案可以应用于广泛的问题类:

- 编译器;
- 软件计算器;
- Web 浏览器;
- 命令解释器;
- 自然语言处理;
- 编码/解码;
- 文件传输。

工资单程序按一般性方式解决了单个问题, 但应用框架是为不同种类的问题提供了单个工作和组织的模式。应用框架的用户相信决定应用程序架构的重要性, 因为应用框架为用户提供了软件组件蓝图, 该蓝图定义了应用程序中主要类之间的预定义关系和工作模式。这听起来似乎是一种限制, 但不必害怕, 因为应用框架广泛定义了具有广泛一般性应用的整个类。提交给框架的应用通常都是意义明确、能准确理解的。而且, 因为它们在最抽象一级为应用建模, 所以应用框架倾向于为开发者提供开发但确定的结构。这暗示着, 如果开发者实现的系统可以划分到这些明确定义的应用之列, 那么, 该系统就可能具有与应用框架提供的预定义模式和关系非常相似的结构。

应用框架使用 C++ 类来构建。在应用框架中最高级的类表示应用的一般性形式。这些类通常都具有非常完备的接口 (Stroustrup, 1991, 455)。当在设计过程中使用应用框架时, 设计工作将得到巨大的简化。设计者可以从一个整体结构推进到一个特定组件。开发者可以自由地使用所需的任何材料, 但构建的架构已经就绪, 已经为程序员提供了边界、限制, 同时还有方向。因为应用框架通常表示定义明确、理解准确的应用, 所以, 它们是为应用提供多线程架构的理想之选。

在一个具有面向对象架构的应用之内, 建议用应用框架解决所有多线程问题。应用框架决定并行和并发有意义的地方。图 1-8 描述了多线程架构的一般性结构。

应用框架的用户在受益于并行的同时应当隐藏它。与域类一起, 多线程应用框架必须正确使用互斥类和 IPC 类来提供支持并发和多进程的软件基础。

在本书中, 我们描述了如何运行 C++ 组件来实现增量多线程编程。演示了 C++ 接口类如何用于封装执行线程、任务及进程间通信的操作系统 API。本书探索和解释了线程化面向对象架构。讲解

了作为经典并发问题（如死锁、无限延迟和数据竞争）面向对象解决方案的应用框架。我们还将看到，基本操作系统功能（例如进程、线程和多任务）支持 C++线程化对象架构的概念。

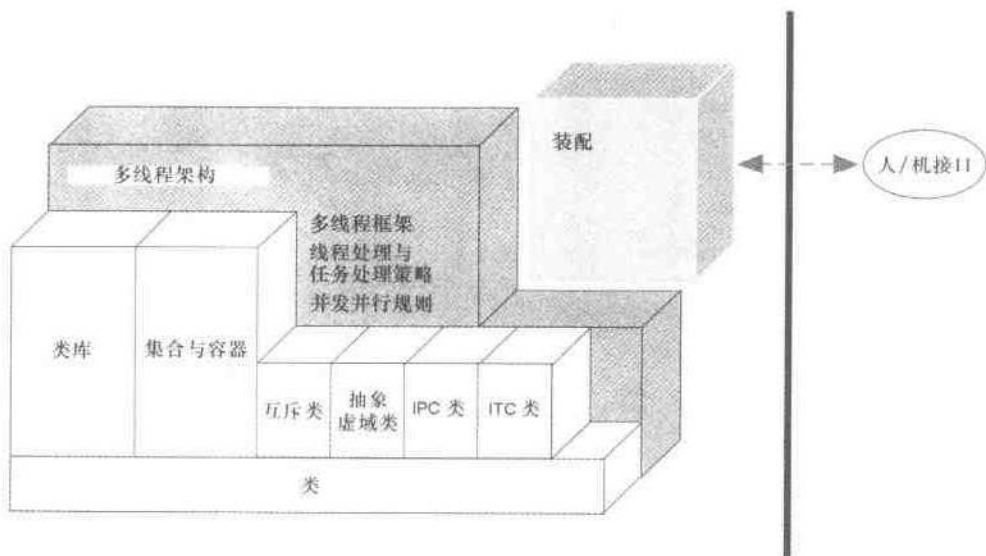


图 1-8 多线程架构概览

进程解剖

...于是，我们赋予地点以意义，于是，像“向上 2 个单位，再越过 1 个单位”这样的操作就会从一种任意运动转换成有语义的操作。

Giving Meaning to Place: Semantic Space

——Alan Wexelblat

为了理解多线程编程，掌握什么是线程（thread）和进程（process）以及它们之间的关系很重要。

2.1 什么是进程

对于本书所讲内容而言，进程就是执行中的一段程序。也就是说，一旦程序被装载到了内存中并准备执行时，它就是一个进程。进程具备文本（text）、数据（data）和堆栈片断（stack segment）以及它自己的资源（resource）。资源可以是文件（file）、对象句柄（object handle）、设备（device）、信号量（semaphores）、互斥量（mutex）、管道（pipe）等等。操作系统管理进程以及它的资源。有大量信息与进程相关。这些信息可以保存在一个称做进程控制块（process control block）或进程信息块（process information block）的结构中。这个结构是进程对于操作系统的表现形式。重要信息定义和描述了对应进程。操作系统需要它来管理进程以及它们的资源。用户通过函数调用或访问数据结构可以使用此结构中包含的信息。

与进程关联的信息保存在进程信息块、环境块以及其他保存进程属性的结构中。函数或应用返回与可被使用进程相关的重要信息。其中一些结构和函数所提供的信息是重叠的。进程信息块中包含的信息有进程名字（或进程 ID）、内存和分配资源的指针、寄存器保存区以及进程的优先权。其他信息可能包含进程运行所在的 CPU（如果系统是一个多处理器系统——即处理器多于一个——的话）以及命令行参数和环境块的指针。进程 ID 是与每个新进程关联的唯一标识。当操

作系统重新获得 CPU 时间时，它需要寄存器保存区所保存的信息来重新启动进程。每个进程都有一个初始优先权。进程优先权表示系统用于决定进程使用处理器顺序的级别数。

进程的属性包含保存在进程信息块中的部分或全部信息，以及与信号、I/O 和虚拟内存操作计数器、进程退出状态和异常/调试端口有关的信息。I/O 计数器和虚拟内存操作计数器是记录 I/O 操作类型和数量以及进程的线程所执行虚拟内存操作的变量。退出状态是进程终止的原因。异常/调试端口为进程间通信渠道。当进程的线程之一产生一个异常或调试进程时，操作系统发送消息给异常/调试端口。

进程的环境是一个由系统和（或）用户定义指针的集合。环境为进程设置缺省信息。环境变量（environment variable）协助定义进程的行为。每个进程对这些变量都有自己的解释。环境变量列表的创建和初始化由用户完成。一些变量具有系统初始化的缺省设置。当进程运行时，它寻找这些变量，并根据它们调整它的行为。这些变量包括用户初始工作目录、查找进程执行文件的路径以及搜索命令的目录列表。用户可以调用函数来创建和初始化特定进程的变量。在某些系统中，它们称做局部变量（local variable）。表 2-1 列出了保存在进程信息块和环境块中的一些信息、其他与进程有关的属性以及提供它们的操作系统。

表 2-1 与进程有关的信息

进程信息块		
OS/2 进程结构	pid	当前进程 ID
	ppid	父进程 ID
	module_handle	程序可执行模块句柄
	command_line	命令行参数的指针
	environment	进程环境块的指针
Win32 进程结构	hProcess	进程句柄
	hThread	线程句柄
	dwProcessID	当前进程 ID
	dwThreadID	线程 ID
环境块		
UNIX	HOME	指向用户初始工作目录的路径
	LOGNAME	与某进程关联的登陆名
	PATH	查找可执行模块的路径前缀
	TERM	输出终端的类型
	TZ	时间区信息

续表

环境块		
	LANG	地区
UNIX	LC_ALL	覆盖地区名
	LC_COLLATE	核对信息的地区名
	LC_TYPE	字符分类的地区名
	LC_MONETARY	货币编辑的地区名
	LC_NUMERIC	数值编辑的地区名
	LC_TIME	地区日期和时间信息名
OS/2	PATH	指向可执行模块的路径
	DPATH	setlocale()所需的可执行模块路径的位置信息, 以及运行时未绑定到可执行模块而且不位于当前目录中时的运行时位置信息
	LIBPATH	DLL 文件的位置
	TMP	临时文件的位置
	TEMPMEM	设置控制是将临时文件在内存中创建, 还是创建为磁盘文件
	COMSPEC	命令解释器的位置
	TZ	时间区信息
Win32	ComSpec	命令解释器的位置
	Path	可执行模块的路径
	Os2LibPath	DLL 文件的位置

一些系统拥有大量函数和应用程序, 它们返回与进程有关的重要有用信息。有些信息也可以在进程的环境或信息块中找到, 如进程 ID 或进程状态。函数返回的其他信息包括: 进程数据和堆栈片断的大小、当前所使用 RAM 大小、CPU 时间的百分量、在最后一分钟进程使用 RAM 的数量、创建和退出进程的时间以及累积执行时间。考虑到不同的操作系统对进程的表示和实现不同, 由操作系统产生的信息也不相同。一些系统产生的信息较少, 而另外一些系统产生的信息比以上提到的还要多。表 2-2 列出了不同操作系统提供的一些应用程序, 它们返回进程的有用信息。

表 2-2

应用程序

环 境	应用程序	描 述
UNIX	ps	生成一个进程状态信息列表
OS/2	pstat	生成一个进程状态信息列表
WinNT	mem/d	生成一个内存驻留进程、程序以及设备驱动程序的列表

2.2 进程状态

对进程可以执行若干操作。进程可以创建和销毁，进程的优先权可以更改，而且进程可以通过一系列进程状态转换。进程状态（process state）是进程某时某刻所处的模式或条件。进程的状态决定了将来的事件以及进程可能进入的状态。表 2-3 列出了几种进程状态及其描述。

表 2-3

进程状态及其描述

状 态	描 述
就绪（ready）	分派器给进程分配了处理器
运行（running）	在处理器上执行的进程
阻塞（blocked）	在外部事件发生前不能运行
备用（standby）	从就绪队列中挑选出在处理器上运行的进程。对每个处理器，某一时刻只能有一个进程处于该状态
空闲/新建（idle/new）	不可运行的新创建进程。这是进程的初始状态
挂起—就绪（suspend-ready）	进程临时从就绪队列或从处理器中删除，因为： <ul style="list-style-type: none"> • 系统操作性能低下，可能会崩溃；问题得到纠正后进程挂起 • 由于结果不精确或不正确，用户挂进程 • 系统可能超负荷，因而挂起进程，直到负荷达到正常水平为止 • 进程为了等待某对象同步化的执行而自愿挂起
挂起—阻塞（suspend-blocked）	当进程等待的事件永不发生或无限延迟时，阻塞进程被挂起
完成/终止（done-terminated）	进程完成执行，而且释放了资源
僵化（zombified）	进程终止，但没有返回父进程它的退出代码。进程接受退出代码前一直处于僵化状态

准备执行的进程处于就绪状态（ready state）。该进程与其他准备执行的进程一起被放在就绪队列中。只有位于就绪队列中的进程才有资格使用处理器。接着，从就绪队列中挑选出一个进程

并在处理器上执行。进程于是就处于运行状态（running state）。如果进程等待某个事件的发生而不能执行，该进程就处于阻塞状态（blocked state）。就绪、运行以及阻塞状态都被看作激活状态（active state），是最常见的实现进程状态。

新建（new）和空闲（idle）状态是新创建进程的初始状态。当进程处于这种状态时，它就准备执行但尚不能运行。新进程只能初始化或准备。当准备一个进程时，它就预备在处理器上执行了，于是进程进入了就绪状态。进程可以越过新建和空闲状态，并被立即当成可运行进程。

在系统中，对于一个处理器只能有一个进程处于备用状态（standby state）。备用状态是就绪状态之后、运行状态之前所处的状态。挂起-就绪状态（suspend-ready state）和挂起-阻塞状态（suspend-blocked state）是两种挂起进程执行的状态。处于这种状态下的进程为非激活。挂起-就绪状态是一种非激活状态，此时，进程临时从就绪队列中删除或解除其运行状态。挂起-阻塞状态是一种进程可以从阻塞状态进入的非激活状态。

当进程完成执行时，它退出系统。所有进程相关信息都被删除，而且地址空间和资源都被释放。这描述的是完成（done）或终止（terminated）进程状态。终止进程可以再次初始化。在进程完成执行之后，但在释放它之前，进程可以进入一种僵化状态（zombified state）。

进程状态转换

进程状态转换就是改变状态，仿佛从一种状态转换到另一种状态一样。当进程从就绪进入运行状态时，进程就发生了状态转换。表 2-4 列出了一些状态转换和它们所发生的状态以及简短的描述。

表 2-4 状态转换及其发生时的状态

转 换	描 述
新建→就绪	新创建进程为可运行而且放置在就绪队列中
就绪→备用	下一个处理器的进程
分派 就绪→运行 备用→运行	分派器给就绪队列中的某进程分配处理器
抢占 备用→就绪	进程被抢占并放回到就绪队列中
时间耗尽 运行→就绪	时间片（时间段）用完
阻塞 运行→阻塞	运行进程由于需要 I/O 或等待着外部事件的发生而放弃处理器

续表

转 换	描 述
唤醒 阻塞→就绪	已发生事件或 I/O
挂起 运行→挂起→就绪 就绪→挂起→就绪 阻塞→挂起-阻塞	进程临时从队列中删除，或者从阻塞状态转换到挂起状态。进程为非激活
恢复 挂起-就绪→就绪 挂起-阻塞→阻塞	进程从挂起-就绪中删除，同时从挂起-阻塞状态转换到激活状态
退出 运行→僵死 运行→终止	进程完成但没有释放 进程完成且释放
重新初始化 终止→新建	进程完成，但重新初始化并再次使用

进程状态变化有多种不同原因。对于就绪、运行和阻塞状态有 4 种可能的转换：

- 分派 (dispatch)；
- 时间耗尽 (timerrunout)；
- 阻塞 (block)；
- 唤醒 (wakeup)。

当就绪队列中的进程提交给处理器，此进程就被分派了。它就会运行一段时间，称做时间片 (quantum)。从运行状态起，进程有可能重新进入就绪状态或进入阻塞状态，这取决于发生的事件。如果时间段用完，进程就会从运行状态转换回到就绪状态。这一状态转换称做时间耗尽 (timerrunout)。它阻止进程独占 (monopolize) 处理器。如果进程需要 I/O 或等待某事件的发生而不能继续，进程就放弃处理器并从运行状态转换到阻塞状态。这种转换叫阻塞。例如图 2-1 所示的例子。

进程 B 提示用户输入一个用于进程 A 的值。首先运行进程 A。它一直处理于运行状态，直到它需要来自进程 B 提示的键盘输入，但因为进程 B 没有运行，所以不能获取输入。当进程 A 到达这一状态点时，进程 A 放弃处理器，并从运行状态转换到阻塞状态。然后，它被放置到其他阻塞进程列表中。进程 A 将处于阻塞状态，而且直到获取了键盘输入前，一直处于阻塞状态。将进程 B 分配给处理器。它提示用户输入数据，于是发出输出请求。它也放弃了处理器控制，并且从运行状态转换到阻塞状态。一旦获取了键盘输入，进程 A 就发生从阻塞状态到就绪状态的另一次状态转换。这种转换称做唤醒 (wakeup)。

图 2-2 显示了运行、就绪以及阻塞进程状态的状态图。

状态图中的节点表示进程的状态，箭头表示进程退出一个状态进入另一个状态时的状态转换。对于一个简单的操作系统，只有3种最常见的实现状态以及4种状态转换。

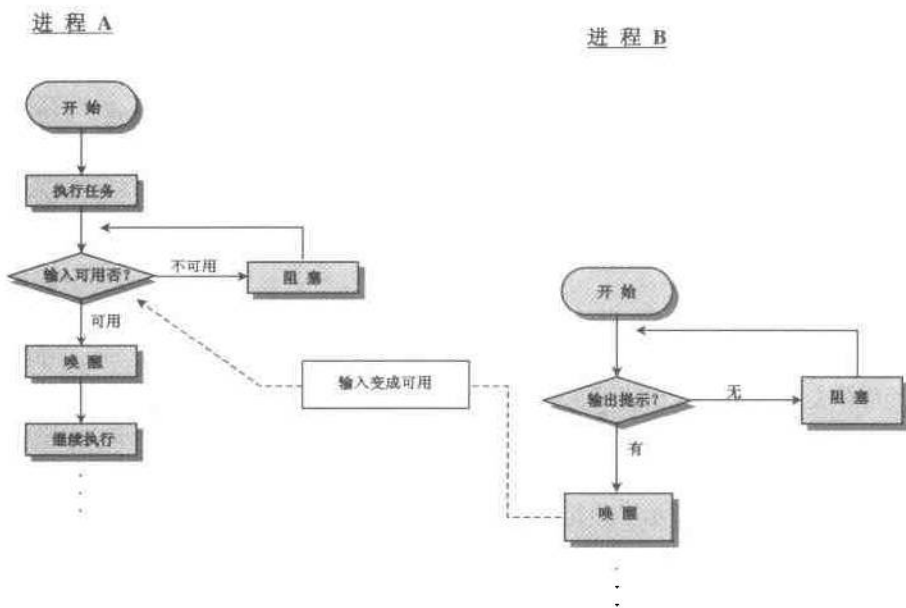


图 2-1 进程 A 和进程 B 从运行状态转换到阻塞状态的流程图。进程 A 在要求键盘输入前一直运行。

要求键盘输入时，进程 A 阻塞。进程 B 运行，然后发出输出提示的请求。进程 B 阻塞。

当获取输入时，进程 A 取消阻塞，继续执行

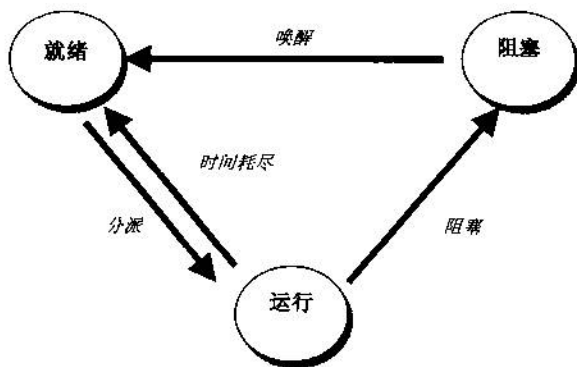


图 2-2 运行、就绪以及阻塞进程状态的状态图

其他状态转换描述了处于挂起状态的进程。进程被挂起的原因有以下几种：

- 系统运行状况不良或中断了。
- 系统可能超载 (overload)，把一些进程放到挂起状态可能会减轻超载。

- 用户可能决定挂起进程，因为它给出了不正确的结果，于是，将进程置入挂起状态，直到问题得到更正为止。
- 为了等待另一个进程同步其执行，进程可能自动进入挂起状态。
- 进程可能请求 I/O 操作，而由于某些原因，对应的资源不可用或者 I/O 操作没有发生，此时进入挂起状态。

就绪、运行和阻塞进程可以被挂起。阻塞进程可以从阻塞转换到挂起-阻塞。如果阻塞进程的资源不可用，则进程可能被挂起，直到资源可用为止。例如，如果进程 A 发出磁盘输入的请求，进程将进入阻塞状态。如果磁盘驱动不能使用，系统就会发现问题，请求的完成就会被延迟。进程 A 直到驱动可用而且完成请求前一直被挂起。一旦资源可用，进程 A 将恢复并重新进入阻塞状态。进程就不再被挂起。

运行进程可能被挂起，从运行状态转换到挂起-就绪状态。用户可能因为进程产生了不正确结果，而决定挂起该进程。如果系统超载并且反应迟钝，进程可能会被挂起。一旦得到了纠正，或系统应用速度恢复了，挂起进程就被重新激活。这一状态转换称做恢复（resume）。挂起进程从挂起状态转换回到就绪或阻塞状态，而决不会直接回到运行状态。如果资源变成可用而且 I/O 请求完成了的话，挂起-阻塞进程可能转换到挂起-就绪，但进程保持非激活。图 2-3 为一个状态图，除了显示常见进程状态外，还显示了挂起状态。

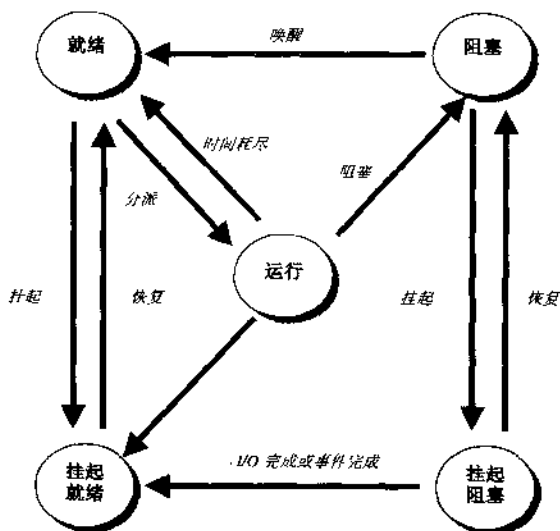


图 2-3 除了显示常见进程状态外，还显示了挂起状态的状态图

一旦进程执行完成，它可能退出系统，并销毁进程。销毁的进程是从内存中删除的进程。它的所有资源都被释放，而且定义进程的所有信息，包括表或列表中的项都被删除。进程执行完成后可能有两种状况。如上所述，进程可能被销毁。这是一个进程的正常终止。终止的进程也可能处于僵化状态。在这种情况下，进程已经完成了执行，但没有释放；与进程有关的信息没有删除。发生这种情况的原因将在讨论进程关系后再作解释。进程也可以被放弃，或某个信号导致进程的提前终止。

图 2-4 和图 2-5 显示了 UNIX 和 Win32 的状态图。

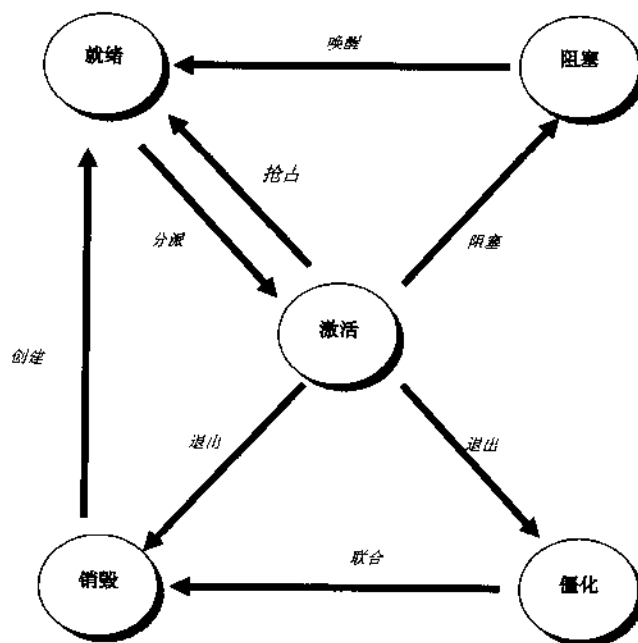


图 2-4 UNIX 中进程状态的状态图

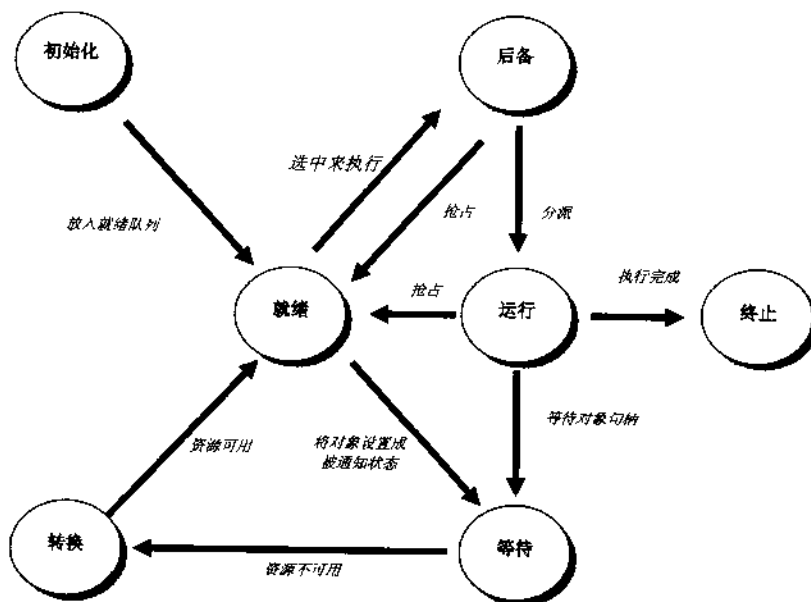


图 2-5 Win32 中进程状态的状态图

这两个系统的状态图有一些区别，包括状态名字、附加状态以及允许的转换。当在 Win32 系统中创建进程时，新进程在将它们放到就绪队列前初始化，然后看作可运行进程。对 UNIX 兼容系统，创建进程时，它就立即被当成可运行。Win32 有一个附加状态，即备用状态 (standby state)，它是下一个使用处理器进程的过渡状态。备用进程可能被抢先，而返回到就绪状态。在 UNIX 系统中，分派进程后，它可能被抢先 (当时间片用完之时)，而且重新进入就绪状态。从运行状态起，它也可以被阻塞，或退出系统。当进程退出系统时，它可能进入僵化状态，或者被销毁。在 Win32 系统中，运行进程也被抢先或终止。它也可以进入一种称做等待 (waiting) 的状态。等待状态就像阻塞和挂起状态的组合一样。当进程等待一个对象或对象组被设置成信号状态时，它就进入此状态。当它退出时，对象处于非信号状态，但在终止时处于信号状态。一旦终止，等待进程则进入就绪状态。进程等待对象的原因有：

- 输入或输出；
- 进程执行同步；
- 进程挂起。

如果进程等待一个 I/O 操作的完成，等待状态就作为一种阻塞状态。对于同步或进程挂起，等待状态作为一种挂起状态。Win32 系统还有一种转换状态，它也用作挂起状态。如果对应等待进程的资源不可用，那么等待进程进入该转换状态。一旦资源变得可用，进程就进入就绪状态。请将它与图 2-2 中的状态图进行比较。在状态图中，如果进程对应资源不可用，它将进入挂起-阻塞状态。资源可用后，进程可能进入阻塞状态再变成一个激活进程，也可能继续挂起，并进入挂起-就绪状态。没有从挂起-阻塞到就绪状态的转换。这说明了不同的系统在进程管理方面有不同的方法。附加状态以及到这些状态的转换在执行中的不同点来捕获进程。

2.3 进程优先权

系统中使用优先权方案 (priority scheme) 来决定就绪进程使用处理器的顺序。进程被分配一个优先类 (priority class) 以及该类中的优先级 (priority level)。类和级从低到高。系统从就绪队列中选择一个最高优先权进程来使用处理器。搜索最高优先权进程时，系统从最高优先类和该类中的最高优先级开始。如果该类中没有这样的进程，则搜索下一个较低优先权的进程。发现目标进程后，就将它提交给处理器执行。如果较高优先权进程变成可用，则运行进程被抢先，即使它的时间段还没有过期也是如此。运行进程被放回就绪队列，将较高优先权的就绪进程提交给处理器。同一类中同一级别的就绪进程按一种轮询 (round-robin) 方式来使用处理器，在更高优先权进程进入系统前，每个进程都有机会使用处理器。进程由外部资源和系统分配优先权。子进程继承了父进程的优先权。之后，它的优先权可以改变。在某些系统中，其他子进程的优先权是可以改变的。如果进程没有分配优先权，则赋予它缺省优先权。

动态和静态优先权

优先权可以为动态，也可以是静态。动态优先权可以变化。进程具有一个初始分配的优先权，此优先权只保持短暂的时间。当系统中发生变化时，可能需要改变进程的优先权来增加系统的响

应速度。动态优先权方案可以帮助较低优先权进程使用处理器。高优先权进程可以降低为较低优先权，而较低优先权进程可以提升优先权。静态优先权不会变化。大部分系统综合运用静态和动态优先权。一个进程可能被分配一个静态高优先权类确保该进程总是使用处理器。例如，与设备控制有关的进程可能被看作一个对时间敏感的进程。迟钝的反应可能导致设备的伤害和毁坏。低优先权类进程可能分配有动态优先权。在某些系统中，抢先进程的优先权值被降低，而等待进程的优先权值被提升（两者都有一个动态优先权分配值）。

2.4 上下文切换

当一个运行进程从处理器中删除，而且系统选择了另一个进程放在处理器上执行时，就发生了上下文切换（context switch）。进程的上下文包括系统需要的进程有关信息以及在以后重新启动进程的环境。上下文包含可执行映像（executable image）、程序计数器（program counter）、寄存器（register）、堆栈（stack）以及用于动态和静态变量的分配内存。系统可能也需要有关进程状态、进程和用户标识、规划和计数信息、优先权以及 I/O 方面的信息。系统还需要知道进程是否等待着某个事件，以及进程占有资源的信息。

在以下情况下会发生上下文切换：

- 进程的时间段用完。
- 一个等待或新建进程准备就绪，而且已经被选择使用处理器。
- 比运行进程较高优先权进程变成就绪状态。
- 运行进程的分配处理器发生了变化。
- 运行进程发出 I/O 请求。

当系统进行上下文切换时，它请求某软件的介入来启动从运行进程到被选中即将运行进程的切换。

2.5 进程关系

当创建进程时，就分配了一个地址空间。文本片断通过一个可执行映像初始化。数据和堆栈片断以及进程的属性都同时被初始化。进程对应的资源也被分配。正如前面所提到的，新创建的进程有一个环境块。进程的环境包括工作目录、标准输入和输出、错误、路径等待。进程成为一个独立的实体。进程可以创建或产生其他进程。原始进程称做父（进程），产生的进程称做子（进程）。子进程可以反过来成为父进程，于是创建了进程的一个树或层次结构。子进程只有一个父进程，但父进程可以有許多子进程。图 2-6 显示了一个可能的进程树结构。

父和子进程描述了它们之间的关系。进程的关系决定了进程的初始化、终止以及父进程对子进程的控制。

在 UNIX 环境中，创建了进程是对其父进程的复制。子进程具有父进程地址空间的一份拷贝。父进程和子进程的数据和堆栈片断是私有的，但它们共享文本片断。子进程和父进程可以访问在两种进程地址空间之外创建的共享内存区域。父进程的一些方面被子进程

继承。子进程可以继承父进程的环境、优先权、规划属性以及父进程的部分资源（包括文件描述符）。子进程也可以继承父进程的属性。在某些系统中，父进程决定哪些能被继承、哪些不能被继承。

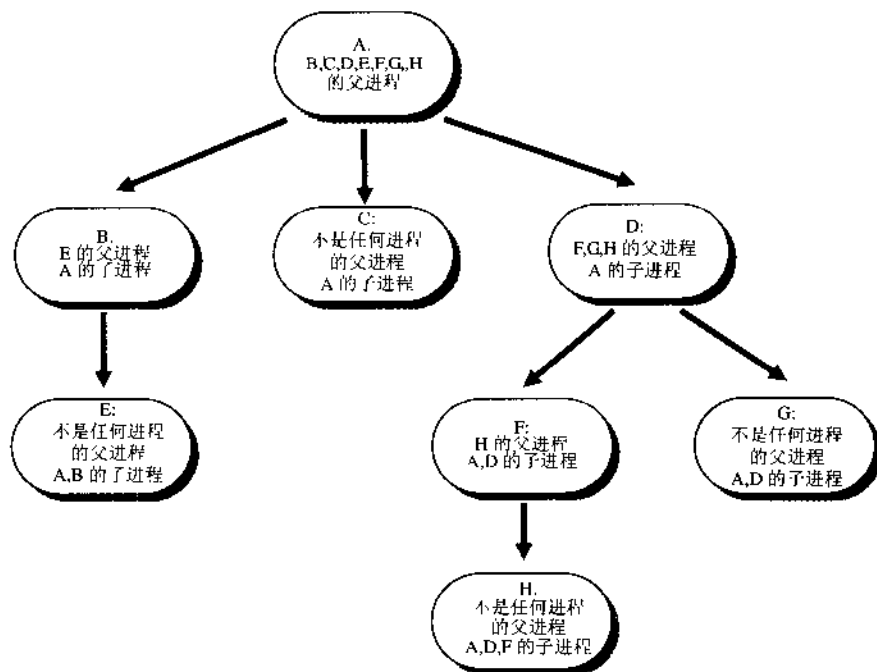


图 2-6 进程树结构示例

不是父进程的所有方面都是可继承的。创建子进程后，子进程和父进程成为两个独立的进程。其程序和堆栈计数器与父进程独立操作。子进程可以更改它的变量值而不影响父进程。对于创建子进程前父进程打开的文件和设备等资源，子进程可以立即访问，但创建了子进程后，子进程就不能访问由父进程分配的任何额外资源。父进程不能访问子进程分配的任何资源。子进程没有从父进程继承进程 ID 或父 ID 这样的属性。子进程必须与父进程和其他进程竞争处理器时间。设置时间信息，如子进程的处理器使用以及创建时间等。

一旦创建子进程，子进程的文本、堆栈以及数据片断可以被另一个可执行映像替换。图 2-7 示例了子进程的创建及其地址空间的重新初始化。

图 2-7 演示了一个从父进程复制的原始子进程，但后来由一个可执行映像替换了子进程的地址空间。尽管地址空间被重新初始化，但子-父关系仍然通过子进程的生存而保持。当子进程终止时，这是影响父进程的重要一点。子进程可以改变它的工作目录和优先权。子进程可以更改环境和其他属性。表 2-5 列出了在 UNIX 环境中重新初始化子进程（对 `exec()` 函数的调用）后保留的属性。

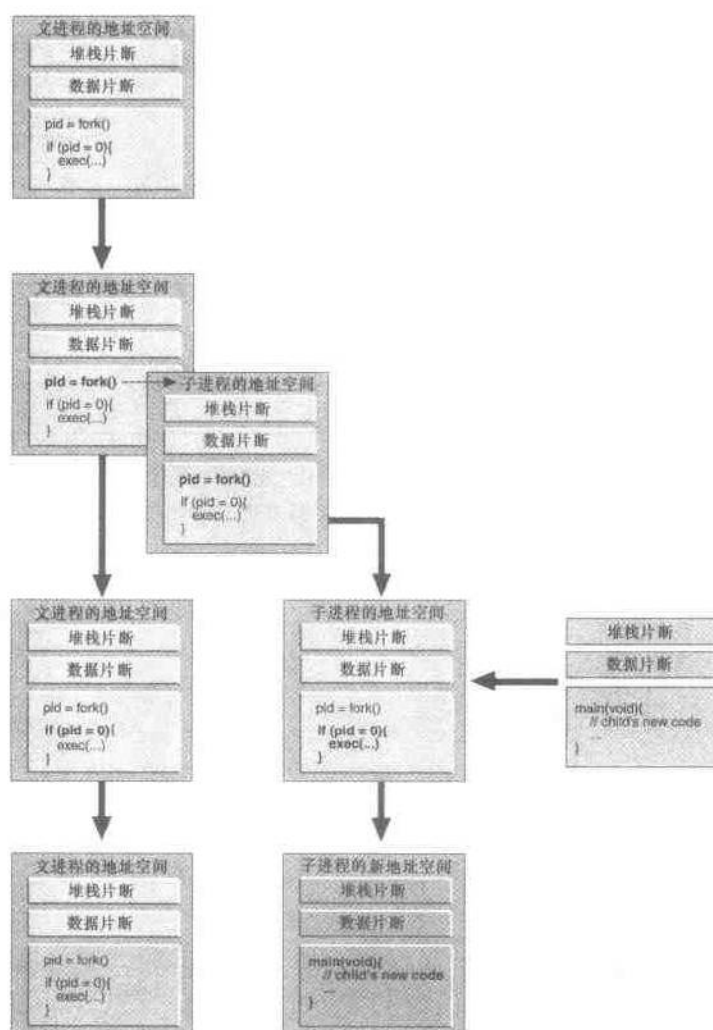


图 2-7 在 UNIX 中创建子进程。子进程首先为父进程的一个拷贝。之后，子进程的地址空间被重新初始化。父和子进程成为两个分离的进程。

表 2-5 在 UNIX 环境中重新初始化子进程后保留的属性

类 别	属 性
标识	进程 ID
	父 ID
	进程组 ID
	实际用户 ID

续表

类 别	属 性
信号	补充组 ID
	进程信号掩码
	待决信号
环境	当前工作目录
	根目录
定时信息	目录为止进程所使用的时间
文件创建权限	文件模式创建掩码

创建的子进程可以不是父进程的拷贝。子进程的地址空间通过一个可执行程序立即初始化。它仍然继承前面所提到的一些属性。图 2-8 显示了这种子进程的创建。

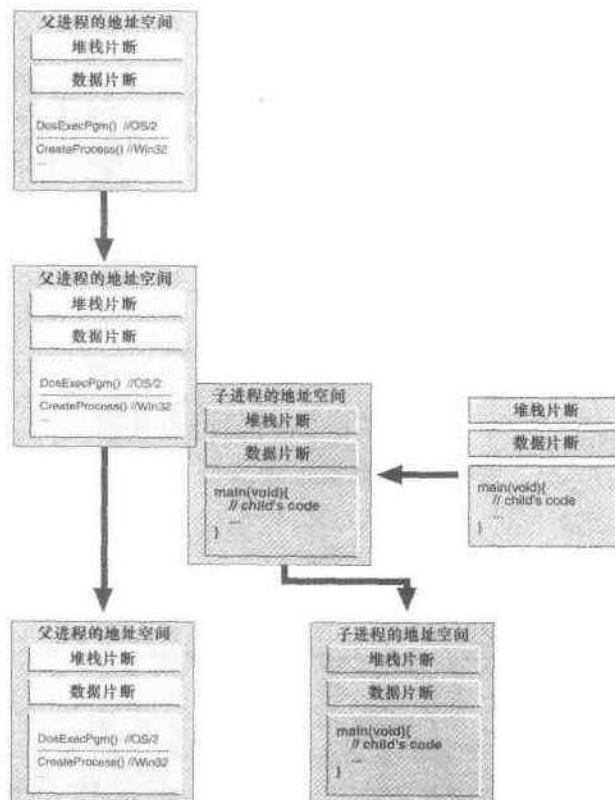


图 2-8 在 OS/2 和 Win32 环境中创建子进程。子进程不是父进程的拷贝。它立即被新的可执行映像初始化

如图 2-8 所示。子进程不是父进程的拷贝，但它立即被可执行新映像初始化。在这种情况下，父-子关系可以通过子进程的生存来维持或不维持。如果不维持，两个进程就是分离的实体，父进程不能对子进程施加任何控制，而且当它终止时对子进程也没有任何责任。不过，对于这两种情况，子进程都可以更改它的环境、优先权和其他一些属性。

2.5.1 进程终止

如果进程间保持一种父-子关系，进程的终止和释放的发生条件可能有多种。在 UNIX 环境中，直到子进程返回退出代码到其父进程中时，它才能得到释放。如果父进程不能接受退出代码，或者由于该代码被忽略，或者由于父进程过期了，那么子进程就不能被释放。进程将一直驻留在内存中，直到它与另一个接受它的退出代码的进程联合为止。例如，进程 B 为进程 A 的子进程，即 A 为父进程。进程 A 在终止前必须等待进程 B 的终止，进程 A 不能在进程 B 之前终止。当进程 B 最后完成执行并试图返回退出代码以警告已经完成执行的父进程时，进程 A 不会接受退出码 (exit code)，因为它已经过期了。此时，进程 B 处于僵化状态。它不能继续执行，因为它已经完成了执行，但不能被释放。它既没有生存，也没有消亡。如果父进程等待着子进程的终止，子进程将可以正常终止。

在其他系统中，进程的终止和释放在以下情况发生：

- 父进程终止，而且所有子进程终止且释放。
- 父进程放弃了进程。
- 内部或外部信号导致终止。

如果父-子关系不再维持，子进程可以终止和释放，且与派生它的进程无关。

2.5.2 同步和异步进程

当父进程挂起执行，直到子进程终止为止，这些进程就具有同步执行 (synchronous execution)。父进程挂起，一直到子进程终止。一旦子进程终止，退出代码返回到父进程，通知父进程子进程已经终止了。当父进程收到退出代码时，父进程就恢复并继续执行。父进程可以设置一个标志来指示进程的同步运行。

父和子进程也可以异步运行。如果父和子进程相互独立运行 (父进程被挂起)，这些进程就是异步执行。图 2-9 显示了同步和异步进程的不同执行。

图 2-9a 显示了父进程与子进程同步运行。创建子进程后，在子进程执行直到完成期间，父进程被挂起。一旦子进程完成，父进程恢复并继续它的执行直到完成。图 2-9b 显示了父进程与子进程异步运行。创建子进程后，两个进程相互独立运行。在这种情况下，它们并发运行，继续执行一直到完成。

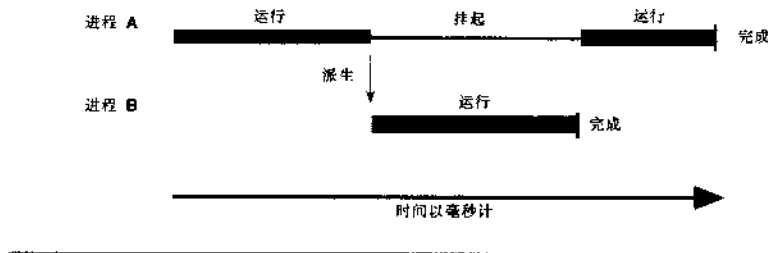
假如一个程序从扫描仪读入一幅图像。在最高级别，扫描图像可以看作 3 个独立的进程：

- 获取图像；
- 显示图像；
- 数据丢失补偿。

图像获取进程通过在图像上移动扫描仪来捕获图像。扫描仪将图像读作数据。显示图像进程取得这部分数据，重新构建图像并发送到屏幕。因为在扫描过程中可能丢失数据，用数据丢失补

偿进程补偿丢失的数据。这些进程创建了一个进程树结构，其中父进程为扫描图像。该进程创建了3个子进程：获取图像、显示图像和数据丢失补偿。

a. A进程和B进程同步运行



b. A进程和B进程异步运行

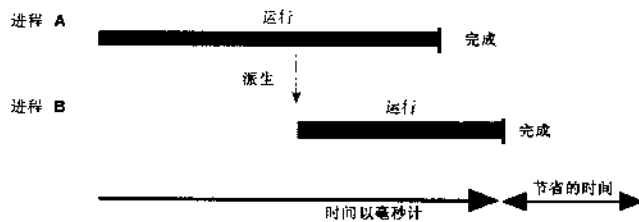


图 2-9 同步和异步进程的不同执行。在图 2-9a 中，父进程与子进程同步运行。在子进程运行直到完成期间，父进程挂起。在图 2-9b 中，父进程与子进程异步运行。创建了进程后，两个进程相互独立运行

扫描图像进程与其子进程同步执行。扫描图像进程在所有子进程终止前挂起。当获得图像数据时，显示图像进程将显示刚才扫描的图像部分。这些进程并发执行，所以是异步的。一旦捕获了所有的图像数据，数据丢失补偿进程就补偿丢失的数据。这一进程在图像数据已经获取而且图像完全显示在屏幕上后执行。数据丢失补偿进程终止后，显示图像进程再次执行。它终止后，其父进程（扫描图像进程）恢复并终止。图 2-10 描述了扫描图像进程以及它的子进程的异步和同步执行。

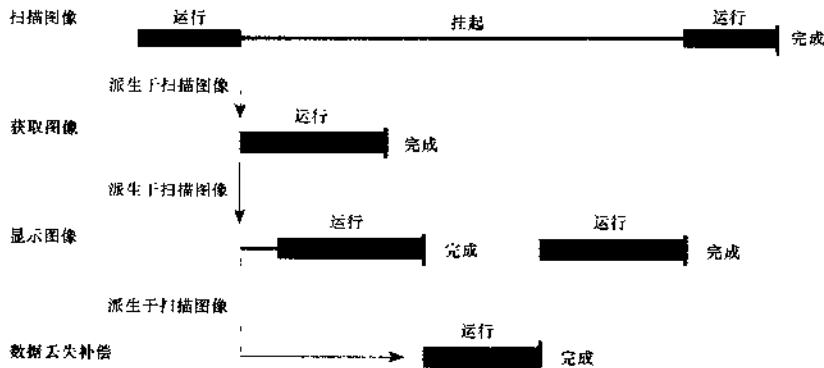


图 2-10 扫描图像进程以及它的子进程的异步和同步执行

如上所讨论的，扫描图像进程派生了所有3个子进程，在所有子进程终止前，它一直保持挂起状态。

异步运行的子进程可以是分离进程（detached process）。分离进程是与其父进程并发运行，但没有从父进程继承任何资源的子进程。父进程终止时，它们不会终止。它们与父进程分离。父进程可以消灭分离的子进程。

2.6 进程映射

一个进程有3个片断：文本片断、堆栈片断以及数据片断。进程的地址空间布局既有物理模型也有逻辑模型。物理模型有关进程如何实际保存在RAM中。逻辑模型在布局的底端有文本片断，后面跟着数据，堆栈片断位于顶部。逻辑布局模型使用虚拟地址空间，它是一套进程可用的虚拟地址。虚拟地址空间映射到RAM物理地址空间上。单个进程的这种映射如图2-11所示。

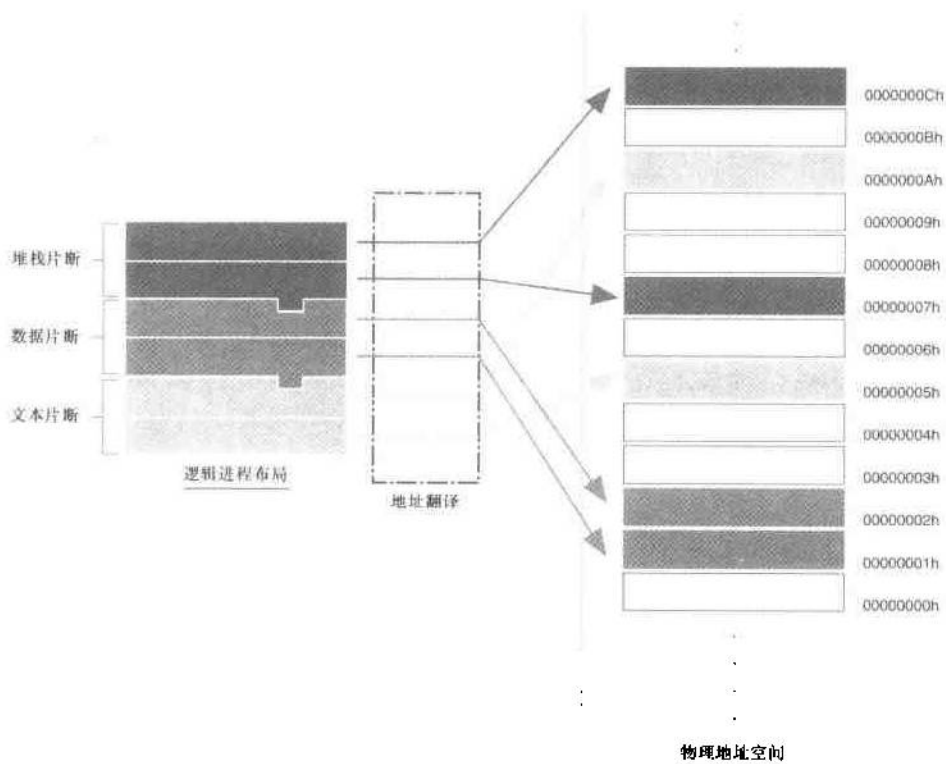


图 2-11 进程虚拟地址空间到 RAM 物理地址空间的映射

片断被分解成数页。当它们被映射时，每页的虚拟地址被翻译成 RAM 中的物理地址。如图 2-11 所示，虚拟地址是相邻的，但物理地址没有必要相邻。使用虚拟存储是因为它解除了虚拟地

址与物理存储之间的联系。

在 UNIX 环境中, 当创建了一个进程时, 就分配了足够的物理存储空间。如果进程派生了另一个进程, 子进程的地址空间或由父进程地址空间的拷贝来初始化, 或者由另一个可执行映像来初始化。如果子进程是父进程的拷贝, 子进程和父进程则共享文本片断。父进程存在于真实的内存中。它具有文本、堆栈以及数据片断。一旦子进程被创建, 文本片断就由两个进程所共享, 但它们有自己的私有堆栈和数据片断拷贝。子进程可以调用系统函数, 导致地址空间被一个新的可执行程序所占据。如果这样, 堆栈和数据片断被释放, 并分配可执行新程序的空间。虽然子进程有一个可执行新程序, 但它是同一个进程, 因此保持着相同的进程 ID 和父 ID。如果进程派生一个非复制品的子进程, 则子进程不会共享文本片断, 而具有完全不同的堆栈和数据片断。

2.7 进程资源

资源 (resource) 是进程执行任务所使用的硬件设备和软件素材。资源包括硬盘、软盘、磁带以及光驱、打印机、显示器、声卡与网卡、文件、程序、数据和库。资源可以是进程在任何给定时间使用的任何东西。它可以是数据源或信息, 或者是显示数据或信息的方式。

系统资源由进程所使用。进程可以使用由操作系统支持和管理的各种不同类型的资源。一些资源被许多进程共享。这样的资源允许多个进程并发访问, 或者在允许另一个进程访问前暂时只允许单个进程的访问。如果两个进程都可以写入变量, 这将导致一个进程覆盖另一个进程使用前的数据。另一方面, 多进程间共享同一个处理器, 但某一时刻只能有一个进程使用处理器。两类资源都被共享, 但前者是并发访问的例子, 后者是连续可重用性的例子。

对其他资源, 只有某个时间由一个进程使用时才有意义。称该进程对资源是排它性访问。当进程需要获取资源时, 它必须首先请求资源, 使用资源, 然后释放资源, 以便让其他进程使用。如果资源不可用, 进程必须等待, 直到可用为止。一旦资源变为可用, 进程就被唤醒。如果进程在特定时间没有使用这一资源, 它就不能被另一个进程使用, 直到该进程释放了资源。一些资源一旦使用后就不能被中断, 而另一些资源可以被中断。共享、非共享、抢占以及非抢占资源的行为取决于资源。基本资源类型有 3 种:

- 硬件;
- 软件;
- 数据。

表 2-6 列出了基本资源、每个资源的示例以及它们的类型。

表 2-6 资源及其类型的示例

资 源		共 享	不 共 享	可 抢 占	不可抢占
硬件	CPU	√		√	
	主内存	√		√	
	打印机	√			√

续表

资 源		共 享	不 共 享	可 抢 占	不可抢占
	监视器		√	√	
	磁盘驱动器	√		√	
	磁带驱动器	√		√	
数据	用户变量	√			
	系统变量	√			
软件	DLL	√			

2.7.1 硬件资源

硬件资源 (hardware resource) 为物理设备。处理器是硬件资源的一个例子。进程使用它来操纵数据和执行程序的指令。主内存和 RAM 是用于数据存储和读取的硬件资源。硬件资源也是 I/O 设备，它为进程提供与自身外部资源间相互发送或接收数据的途径。

处理器是一种可以被多个进程所共享的硬件资源。在单处理器多程序系统中，处理器被多个进程共享。为了模拟同步执行，系统允许每个进程在短时间内使用处理器。进程处于就绪状态后，就向处理器发出请求。系统从就绪队列中选择下一个进程来使用处理器。进程使用处理器一段时间，然后进程必须释放处理器。处理器是一种抢占资源，一旦对应进程的执行时间段用完或进程发出了 I/O 请求，该资源就会中断。

RAM 是另一个硬件资源的例子，它被多个进程共享，而且可抢占。进程的堆栈、数据和文本片断被分成数页，并映射到物理内存。当进程没有使用时，这些页可能被交换到第二存储空间，另一个进程交换进入，并占据自由空间。主内存被多个进程同时共享，因为多个进程有一套位于主内存中的页。在某一时刻，只能有一个进程占据特定范围的内存位置。当一个进程页交换出去，另一个进程页占据此空间时，主内存被抢占。

打印机是一种非抢占资源。许多资源可以给打印机发送任务。这些任务保存在打印机队列中等待打印。一旦打印机开始处理其中的任务，就不能被其他等待任务中断。此时的任务是排它性的。打印机可以被一个发送终止使用打印机当前任务信号的进程所中断。

对话

屏幕、键盘以及鼠标都是为进程提供 I/O 的资源。这些 I/O 资源可以被多个进程通过创建对话 (session) 来共享。对话是一个从逻辑上分离的键盘、鼠标、屏幕以及与这些资源关联的进程组单元。屏幕、键盘和鼠标称做虚拟控制台 (virtual console)，因为它们为这些与对话关联的进程提供了一个虚拟计算机。对话允许不同的进程组共享屏幕、键盘和鼠标，而不会互相干扰。对话可以表现为一个窗口或全屏。与对话关联的进程将输出发送到屏幕，用户也可以通过键盘或鼠标为进程提供输入。用户可以从一个对话自由移到另一个对话。每个对话屏幕不会被中断。

2.7.2 数据资源

数据是一种资源。进程可以使用和操纵数据（比如文件、对象和句柄）、系统数据（比如环境变量）以及全局定义变量（比如信号量和互斥量）。

打开一个文件的同时，可以限制另一个进程访问该文件的类型。可以打开一个文件，允许或不允许其他进程访问它。文件可以按仅提供读访问、并禁止写访问的方式打开，也可以为另一个进程提供读/写两种访问权限。在进程间保持父-子关系的系统中，子进程继承了父进程的资源。子进程在创建之时就可以立即访问父进程的打开文件。子进程还继承了文件指针。当父进程或其他子进程访问此文件时，推进文件指针，下一次访问此文件时，保持此偏移值。这些文件可以表示物理设备。

共享内存可以定义为在多个进程间可见的内存。进程的虚拟空间映射到相同的物理内存位置。虚拟空间可以设计为只读或读/写。如果共享内存可以更改，假设为共享内存缓冲，虚拟空间被设计为读/写。

为了防止数据破坏必须小心谨慎。访问共享内存应当同时避免覆盖数据。例如，两个进程可以共享一个全局变量。一个进程搜索一张磁盘上所有的目录，查找指定文件扩展名的所有文件，并将文件保存在一个列表中。其他进程从搜索列表中的每个文件，查找某个字符串。全局变量在列表中保存文件数，这是字搜索进程没有搜索到的。当文件搜索进程在列表中添加文件时就增加全局变量。字搜索进程从此列表中搜索到一个文件时则减小全局变量。两个进程共享该全局变量。访问此变量必须同步化，防止文件搜索进程在字搜索进程试图减小变量时增加变量。危险情形是，文件搜索进程可以连续增加全局变量，阻止字搜索进程在适当的时候减小变量，反之亦然。如果允许这样的话，当进程完成执行后读取变量时，似乎其中一个进程没有完成它的任务。可以使用信号量（semaphore）、互斥量（mutex）、临界区（critical section）等来实现同步。内存也可以通过存在于多个进程的虚拟地址空间中而被共享。环境变量就是这样。所有的进程都具有这些变量的拷贝。

2.7.3 软件资源

软件资源可以是其他进程或动态链接库（dynamic link library）。库可以是针对进程的一个语句、子程序等的源代码。共享软件资源的进程共享程序的一个拷贝，但拥有数据的单独拷贝。程序的进程虚拟地址被映射到相同的物理内存位置（如图 2-12 所示）。

使用时不发生变化的程序可以被多个进程使用。使用时可能发生变化的程序在进程每次使用它时重新初始化。程序一次只能被一个进程所使用。

2.7.4 优先权与资源

为了让系统区分各个进程，以确保足够的系统操作性能，进程可以给进程任意分配优先权。可以分配和更改优先权，以确保所有的进程充分利用处理器，让进程可以进入系统，同时也可以退出系统。也可以基于进程使用的资源分配进程优先权。通信密集进程需要相对高的优先权。进

程不能使用处理器可能导致数据的丢失。使用或控制时间敏感设备的进程甚至需要更高的优先级。在这种情况下，进程不能使用处理器可能导致设备的损害。

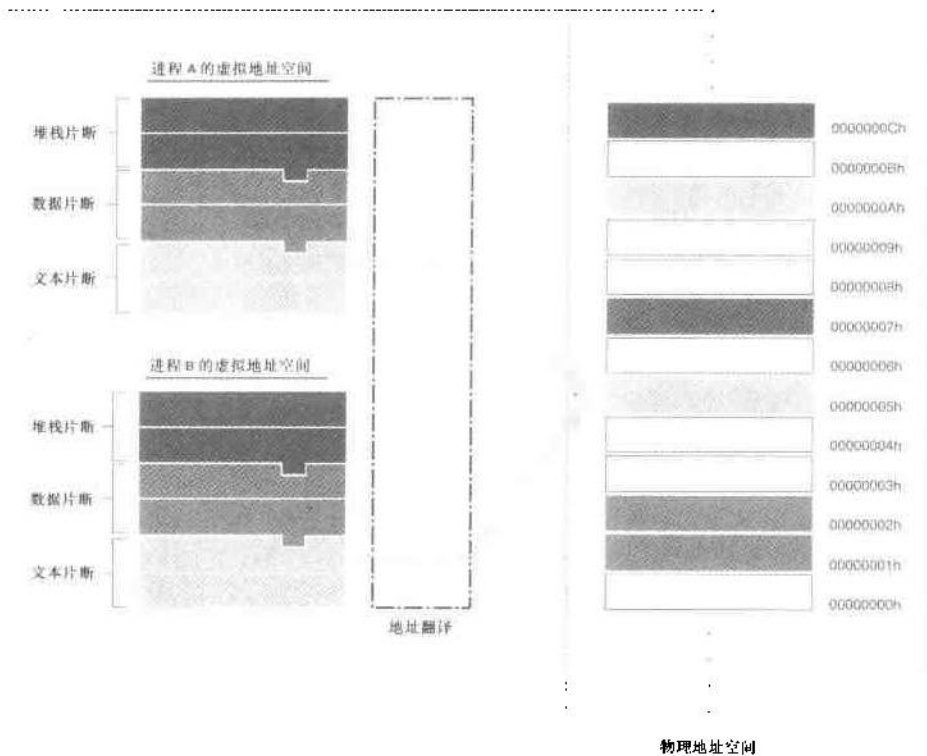


图 2-12 两个进程的虚拟空间的映射，其中一个虚拟地址页被映射到相同的物理内存位置

轻量级进程：线程

融合在两个（或多个）最初不相关的符号参与“接合激活”时发生。它们相互间传递信息如此紧密，因此被捆绑在一起。之后，可以把这种结合体当成单个符号一样来联络。

Godel, Escher, Bach: An Eternal Golden Braid

——Douglas Hofstadter

线程（thread）是一种轻量级进程（lightweight process）。与进程相比，线程给操作系统带来的创建、维护和管理负担要轻，因为与线程相关的信息非常少。较轻的负担意味着线程的代价或开销比进程少。当从处理器删除一个进程并激活另一个进程时，就发生上下文切换。为了发生上下文切换，操作系统必须启动和重新启动每个进程所需的信息。这意味着必须保存描述进程存在状态的所有相关信息，在进程再次激活时，就可以从离开的地点继续执行。所需信息包括可执行程序、堆栈以及静态与动态分配变量内存的指针。寄存器组包含执行下一条指令的指针这样的信息。重新任命进程时需要进程的状态和优先权。进程由于 I/O 请求是挂起，还是阻塞？程序的 I/O 状态将被保存。同时保存进程的优先权，还有规划信息、内存管理信息以及计数信息，并作为进程上下文的一部分。进程是否等待着某事件或信号？也需要与事件或信号相关的信息。进程上下文中包含的大部分信息都与描述进程的地址空间有关，其他信息则与进程所拥有的资源有关。进程需要文件描述器和读/写指针来继续使用资源。

进程上下文使用许多系统资源来跟踪所有这些信息，进程上下文切换才能从容进行。线程也有上下文。当线程被抢先时，必定发生线程间的上下文切换。线程没有地址空间。线程包含在进程的地址空间中。所以，重新任命进程所需的大部分信息在重新任命线程时并不需要。事实上，线程上下文只包含一个堆栈、一个寄存器组和一个优先权。寄存器组包含程序或指令指针以及堆栈指针。线程文本包含在它的进程的文本片断中。进程拥有的所有资源都属于线程。所以，与资源相关的所有信息不是线程上下文的一部分。其他信息，如规划、计数等等都由进程所定义。表 3-1 对比了进程上下文与线程上下文。

表 3-1

进程上下文与线程上下文对比

上下文内容	进 程	线 程
可执行文件的指针	◆	
堆栈	◆	◆
内存（数据片断和堆）	◆	
状态	◆	◆
优先权	◆	◆
程序 I/O 状态	◆	
授予权限	◆	
规划信息	◆	
帐务信息	◆	
内存管理信息	◆	
资源相关信息 <ul style="list-style-type: none"> • 文件描述符 • 读/写指针 	◆	
事件和信号相关信息	◆	
寄存器组 <ul style="list-style-type: none"> • 堆栈指针 • 指令计数器等 	◆	◆

进程的数据片断被它的线程共享。线程可以读和写进程的内存地址，而且进程可以访问数据。当进程写入内存时，线程可以访问这些数据。线程的堆栈包含在进程的堆栈片断中。线程可以在它的进程中创建另一个线程。一个进程中的所有线程称做同位体（peer）。所有的线程共享进程的资源 and 内存。线程不拥有任何资源。由任何线程创建的任何资源都被它的同位体共享。线程也可以在进程中挂起、恢复和终止其他线程。

进程与它的线程几乎共享一切，包括它的资源和环境变量。数据、文本片断以及所有资源都与进程相关，而不是与线程相关。线程是进程的租借者。线程发挥作用所需的一切都由进程提供和定义。所有线程都有一个线程 ID、定义线程状态的寄存器组、优先权以及堆栈。这赋予每个线程一个单独的身份。图 3-1 显示了一个进程的逻辑布局与一个线程的逻辑布局对比的情况。



图 3-1 进程的逻辑布局与线程的逻辑布局对比

3.1 多线程处理

多线程进程拥有的线程多于一个，每个线程都执行进程的程序代码。指令序列称做执行线程（thread of execution）。如果进程只有一个线程，则称做主线程（main thread 或 primary thread），它只有一个控制流程。进程的指令可以分解或分组成子任务（典型情况是函数），这些子任务可以异步执行。指令序列由表现为一个线程。线程可以执行单个函数或一组函数。每个线程的程序计数器将跟踪执行的下一条指令。每个线程允许进程的函数独立执行，而与进程的主控制流程无关。

可以用多线程实现的一个典型进程例子是执行自然语言处理（natural language processing, NLP），NLP 试图赋予计算机理解以人类语言书写或发出的命令的能力。组成命令所需的信息从语言句子中提取，然后执行某个动作。NLP 的核心是称做解析器（parser）的代码部分。它的目的是逐字读取和分析句子。如果让解析器从一个句子或字符串中提取一个命令，该字符串首先必须是有效的。这包含限制可被接受的语法和语言。限制语法意味着限制句子的类型以及句子的结构。例如，通过下面结构只接受陈述句：主语+谓语+宾语，与疑问句刚好相反。限制语言意味着限制词汇。这可能意味着将可接受的单词或符号，以及语音或符号在命令中发挥作用的目保存在一个数据库中。因为句子不仅仅由单词组成（空格、标点、运算符等等），所以使用的是符号，而不是单词。如果句子有效，就从字符串中提取不必要的符号。不必要的符号是不存在于数据库中的符号。如果发现了这样的符号，就舍弃它们并构建一个只由有效符号组成的新字符串。解析器然后在有效字符串中搜索命令部分。再次搜索数据库来判断每个符号在命令中所起的作用。命令部分可能有多个。每个部分相互独立进行搜索。这些搜索可以异步执行。

这种类型解析器的一个例子是一个解析句子提取一个即将执行的 MCI 命令的进程。MCI 代表媒体控制接口（media control interface）。通过使用高级别命令，MCI 能控制不同的多媒体设备。它关于内存分配、跟踪设备句柄以及其他所有低级别细节。解析器允许用户使用自然语言来命令音频、视频和 MIDI 音序器设备。进程将解析包含一条 MCI 命令的字符串，并执行该命令。MCI 命令符号从字符串中提取，并舍弃所有不必要的符号。重新装配符号成一条可执行的 MCI 命令。

这一进程的一般算法如下所示：

验证字符串

从字符串中提取噪音

符号化字符串

获取命令、文件名以及设备符号

构建 MCI 命令并执行

命令解析器由以下子任务组成：

- 验证字符串；
- 提取噪音；
- 符号化字符串；
- 执行。

符号化字符串子任务调用：

- 获得命令符号；
- 获得设备名字符号；
- 获得文件名符号。

验证字符串判断字符串是否有效。如果字符串包含 MCI “谓语”（动词）的其中之一就视为有效：打开、播放或关闭。提取噪音是从字符串中提取 MCI 符号，并舍弃不必要的单词。将字符串中的所有符号与保存在列表中的有效符号进行比较。如果字符串的符号与列表中的符号不匹配，则舍弃之。“符号化字符串”调用“获取命令符号”、“获取设备名字符号”以及“获取文件名字符号”，它搜索字符串寻找命令、设备名字以及文件名字符号。每个符号都有一种定义了它在命令中用法的类型：命令、文件、设备或别名。每个函数从字符串取出一个符号，搜索列表，然后检查它是不是正在搜索的符号类型。一旦发现了目标符号，它不再继续搜索，因为它假定该符号只在字符串出现一次。每个符号都保存在一个变量中。“获取命令符号”搜索 MCI 命令（打开、播放或关闭）。“获取设备名字符号”搜索表示媒体设备名字及其别名的符号。“获取文件名字符号”搜索表示文件和文件扩展名的符号。“执行”装配命令并执行此命令。进程执行有 3 条命令。以下是每条命令的模板：

- open+设备名字+alias+别名
- play+文件名
- close+设备别名

命令的一般模板为：

- command[[设备]alias[别名]][文件名]

如果给定字符串：

“Could you play the file song.wav for me?”

“命令解析器”首先判断此字符串是有效的，因为它包含动词 play。然后从字符串提取所有不需要的符号。新字符串只包含有效的 MCI 符号：play song.wav。“符号化字符串”调用“获取命令符号”、“获取文件名字符号”以及“获取设备名字符号”，它们扫描命令、设备名字以及文件名字的新字符串，并将它们保存在内存中。如果字符串不包含符号，则保存一个 null 字符。“获取文件名字符号”的代码如下所示：

```

getToken();
if(findTokenType(token)=FileName){
    strcpy(MciFile, token);
    return(1);
}
strcpy(MciFile, "");
return(0);
}

```

其他函数的代码实质上与之相同。这一个进程容易应用于多线程编程。多线程应用程序指那些可以受益于子任务异步执行的应用程序。如前面所提到的，“命令解析器”由以下子任务组成：验证字符串、提取噪音、符号化字符串以及执行。每个子任务可以表示一个线程。一些线程同步执行，一些线程可以异步执行。

字符串必须验证，不必要符号从字符串中提取，并构建一个由有效符号组成的新字符串。符号化线程创建了搜索命令、文件名和设备名字符号的线程。符号化线程与它创建的线程同步执行。在所有的线程执行之前，它将挂起其执行。“获取文件名字符号”、“获取命令符号”以及“获取设备名字符号”异步执行。这些任务相互独立。字符串搜索某条命令。如果字符串不包含命令（打开、播放或关闭），就没有必要进行任何符号提取或雾里看花文件名字或设备名字。这些将导致不必要的处理过程。命令可能没有构建或执行。如果字符串有效，而且进行了提取，命令、文件及设备名字符号的查找可以异步执行。图3-2演示了“命令解析器”进程的同步和异步线程执行。

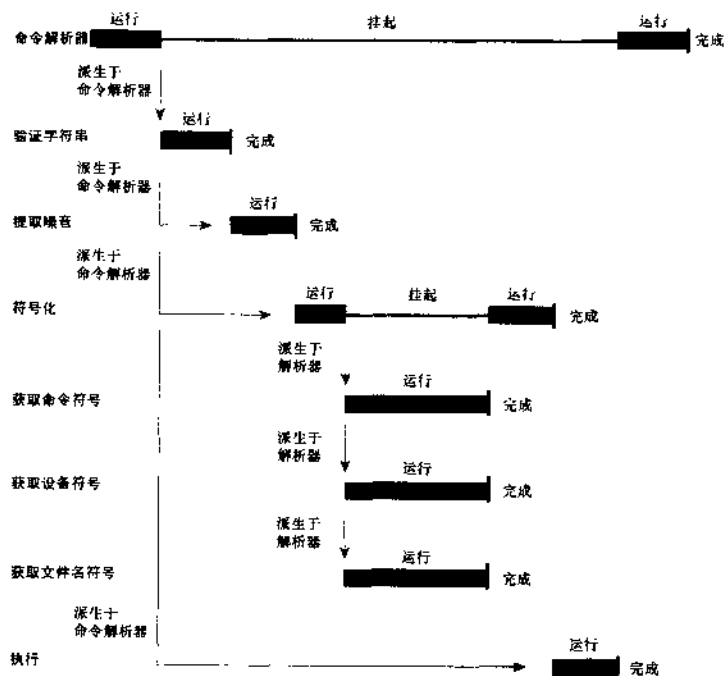


图3-2 “命令解析器”进程的同步和异步执行。解析器与进程的其他线程同步执行。

“获取命令符号”、“获取设备符号”、“获取文件名字符号”线程异步执行

3.2 线程与进程的相似之处

线程在很多地方与进程相似。线程和进程都有 ID、寄存器组、状态以及优先权。它们与之关联的信息块，称做线程块（thread block）和进程信息块（process information block）。线程和子进程共享父进程的资源。进程打开的资源，线程和父进程的子进程可以立即访问。不需要额外的初始化或准备。创建后，线程和进程是与父或创建者独立的实体。父和子进程、线程和创建者一起竞争使用处理器。进程的创建者或线程对它的创建施加一些控制。创建者可以销毁、挂起、恢复和更改进程或线程的优先权。线程和进程可以改变自己的属性和创建新的资源。它们不能访问其他进程资源，或其他进程的线程的资源。

3.3 线程与进程的不同之处

尽管线程与进程有许多地方相似，但在许多重要的地方，两者却不相同。因为线程为轻量级进程，所以线程与进程的主要不同之处为：线程没有自己的地址空间（address space）。如果进程创建多个线程，所有的线程都将包含在它的地址空间中。这就是为什么它们如此容易共享资源，以及为什么可以访问进程内存的原因。进程有自己的地址空间。子进程只有数据片断的拷贝。对变量的修改不会影响父进程的数据。为了让父进程和子进程共享内存，必须创建一个共享内存区域。父和子进程必须使用进程间通信机制（如管道）在两者间通信和传递数据。

通过读取和写入进程变量，同一进程的线程可以传递和使用它们处理的数据。

对其他进程的控制限制于父-子关系。子进程对两者之间无控制。进程内的线程被看作同位体，而且处于相等的级别。不管是哪一个线程创建了哪一个线程，进程内的任何线程都可以销毁、挂起、恢复或更改其他线程的优先权。线程也要对整个进程施加控制。进程内的任何线程可以通过销毁主线程来销毁该进程。销毁主线程将导致该进程的所有线程销毁。对主线程的修改可能影响进程的所有线程。对进程优先权的更改将改变进程内继承了优先权但没有作修改的所有线程的优先权。下面总结了线程与进程间的相似与不同之处。

进程与线程间的相似之处：

- 进程与线程都有 ID、寄存器组、状态以及优先权。
- 进程与线程都有信息块。
- 进程与线程都与父进程共享资源。
- 进程与线程在创建后都是独立的实体。
- 进程与线程创建者对它们施加控制。
- 进程与线程都可以在创建后更改属性，创建新的资源。
- 进程与线程都不能直接访问其他无关进程或线程的资源。

进程与线程间的不同之处：

- 进程有一个地址空间；线程没有地址空间。
- 父和子进程必须使用进程间通信机制，同一进程的线程通过读取和写入数据到进程变量来通信。

- 子进程对任何其他子进程不施加控制。进程的线程被看作同位体，并对进程的其他线程施加控制。
- 子进程不能对父进程施加控制。进程的所有线程都可以对主线程施加控制，并因此影响到整个进程。

3.4 线程的优点

与使用多进程相比，使用多线程来管理一个应用程序的子任务有各种不同的优点。当创建一个进程时，系统就创建了一个主线程。主线程为该进程的执行业务线程。这可能是实施进程功能的唯一所需线程，但是如果进程有许多并发子任务，多线程就可以提供子任务的异步执行。这种方式对上下文切换所花的开销较少。并发多进程（一个进程一个线程）需要上下文切换的较大开销。对每一个线程，必定发生一个进程上下文切换。通过一进程多线程，如果来自另一个不同进程的激活线程为下一个将要使用处理器的线程，只有此时才会发生进程上下文切换。较少的开销意味着使用较少的系统资源，而且上下文切换时间较少。

多线程可以增加应用程序的吞吐能力。通过一个线程，一个 I/O 请求就可以停止整个进程。通过多线程，当一个线程等待 I/O 请求时，应用程序可以继续执行。当一个线程阻塞，另一个线程可以执行。整个应用程序不能等待每个 I/O 请求的完成。其他不依赖于阻塞线程的任务可以执行。

线程并不需要子任务间的特殊通信机制。线程可以轻易地和直接地在任务间传递与接收数据。这也节省了系统资源，如果使用多进程，程序在启动、维持特殊通信机制时必须使用这些系统资源。线程通过使用进程内所有线程共享的内存来通信。进程也可以通过共享内存来通信。进程有单独的地址空间，共享内存位于两个进程地址空间之外。图 3-3 显示了进程和线程间的通信。

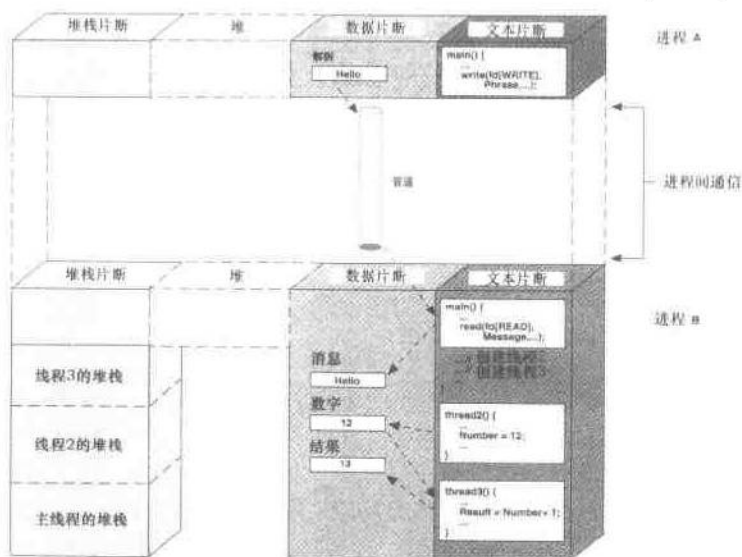


图 3-3 进程和线程间的通信。进程使用共享内存通信。它们有单独的地址空间，而且共享内存位于两个进程的地址空间之外。线程直接通过变量通信

3.5 线程的缺点

线程易于访问进程的内存有其优点，也有其缺点。虽然线程间不需要特殊的机制来通信，如通过进程间的管道，但线程需要同步并发访问内存。线程 A 试图写内存地址，同时，线程 B 正写入同样的地址。线程 C 的任务是读取保存在内存地址的值，并应用于计算中。在线程 C 有机会读取值前，线程 B 覆盖了线程 A 的数据值。线程需要同步化，在线程 B 覆盖值之前，让线程 C 可以读取线程 A 保存在内存地址中的值。需要使用同步来阻止任何一个线程在使用目标数据前覆盖该值。当发生写时，必须发生读。

进程是孤立的。如果任务分解为不同的进程，如果一个进程创建了不良数据，这一数据只限于该进程。如果任务分解成多线程，一个线程可能产生影响其他线程的不良数据。如果线程在处理中使用了该数据，不良数据可能影响进程的所有数据。进程可能由于一个访问违规而导致进程终止。如果违规不是太坏，所有其他进程将继续执行。数据错误只限于单个进程。线程导致的错误比进程产生的错误付出的代价更大。线程导致的访问违规可能导致整个进程的终止。线程可能产生影响所有线程整个内存空间的数据错误。

进程更独立。一个应用程序可以将任务分成多个进程。这些进程打包成模块，这些模块可以用于其他应用程序中。其他应用程序中的模块可以导入新的应用程序。线程依赖于它所属的进程，并且不能退出到创建它的进程之外。进程可以保护资源不被其他进程随意访问。线程与进程中的所有其他线程共享资源。

3.6 线程类型

线程有不同的类型。在 Steve Kleiman、Devang Shah 以及 Bart Smaalders 编著的 *Programming with Threads* 一书中，对线程编程策略的不同类型作了讨论，称做线程范例（thread paradigm），它实质上是应用程序使用的线程编程途径。此范例定义了线程如何完成它们执行的任务。线程策略有：

- 休眠（sleeper）和单步（one-shot）；
- 先占工作（anticipating work）；
- 延迟工作（deferring work）。

3.6.1 休眠（sleeper）和单步（one-shot）

休眠线程在系统中某事件发生前一直挂起。当事件发生时，休眠线程得到恢复，并执行任务。执行任务后，线程为非激活，直到事件再次发生。单步线程实际上就是休眠线程，但它们只执行任务一次后就终止。休眠线程和单步线程也可以称做监视线程（monitoring thread）。它们监视应用程序的某些方面，等待事件的发生，然后作为反应采取一些相应的行为。监视线程的一个例子是监视 COM 端口（COM port）。当端口激活时，线程挂起，但当 COM 端口非激活时，线程苏醒并监视停止工作的过程。如果停止工作超过 5000 毫秒时，线程关闭通信端口。当线程关闭 COM 端口时，它就不再监视任何对象，于是可以终止了。休眠线程重复某个动作。休眠线程的一个例

子是监视打印队列。当某个任务放入队列中时，线程将此任务发送到打印机。队列中可能有多个任务，所以线程被挂起，直到它必须再次执行为止。

3.6.2 先占工作

某个任务被请求执行前执行时就使用先占工作策略（anticipating-work strategy）。对情况进行估计，并先占执行一些动作。执行动作可能没有得到请求。如果被请求，它已经完成了执行。先占工作策略促进使用永不等待规则（never-wait rule）。永不等待规则的目的是阻止非激活。这一规则表明预先执行一个可能不被使用的任务比处于空闲要好。例如：

```
d=a+a
if d>10 then a=5
x=5*a
```

根据第一条语句中计算的 d 值，第二条语句改变 a 的值。如果 a 的值没有改变，第一条和第三条语句可以同时执行。如果 a 的值改变了，第一条和第三条语句就不能同时执行。第二条语句更改 a 的值，然后第三条语句才能执行。永不等待规则认为，第三条语句应该与第一条语句同时执行，因为存在 a 的值发生改变的可能性。如果每条语句都由一个线程执行——线程 A 执行语句 1、线程 B 执行语句 2、线程 C 执行语句 3——那么，线程 A 和线程 C 异步执行，然后执行线程 B。如果线程 B 改变 a 的值，则重新执行线程 C。如果没有改变 a 的值，则该计算已经执行，这就节省了计算时间。

3.6.3 延迟工作

线程可以将任务延迟或推迟到将来某个时间执行的另一个线程。进行延迟的线程可能正执行一些重要的任务，不能执行另一项任务。此时，线程可以创建另一个线程来执行任务，或者将任务交给一些已经存在的线程。可以在后台把一项延迟任务交给分离线程执行。该任务可以在晚些时候执行，因为它不是关键性任务。这一线程的优先权可以为低级别。表 3-2 列出了线程类型及其简短描述。

表 3-2 线程类型及其简短描述

线程类型	描 述
休眠	直到某事件发生前线程一直挂起。它执行任务后再次挂起，等待事件的再次发生
单步	直到某事件发生前线程一直挂起。它执行任务后终止
先占任务	线程在被要求前已在执行某任务。这个动作可能被请求了，也可能没有被请求
延迟任务	线程将某任务推迟到另一个线程在将来的某点执行该任务

3.7 线程相关信息

与进程一样，也存在与每个线程相关的信息。大部分定义线程行为的信息由线程的进程定义。

线程相关信息保存在数据结构中，通过操作系统提供的函数返回。一些线程相关信息包含在一个称做线程信息块（thread information block）的结构中，它在创建线程时得以创建。这一结构可以包含如下信息：指向线程异常处理器头的指针、线程的堆栈基和大小、线程 ID 以及调用线程的优先权。在 Win32 环境中，线程具有一个模拟线程环境的信息结构。这个结构唤醒标志、线程的消息队列、虚拟化输入队列以及其他变量。唤醒标志在挂起线程接收到一条必须处理的消息时由系统设置。为了处理此消息，CPU 对线程安排规划。由操作系统提供的函数可以返回创建和退出时间，指定线程的寄存器数据。

POSIX 环境有一个针对线程的属性对象。这个对象是线程属性的一个封装体。属性对象可以与一个线程关联，也可以与多个线程关联。当使用属性对象时，它就是一个线程或线程组的剖面。属性定义线程的行为。使用这个属性对象的所有线程继承由属性对象定义的所有属性。当属性对象的属性改变时，与属性对象关联的所有线程都将采用新的属性。一旦创建和初始化了对象，它就可以在线程创建函数的调用中重复引用。这将创建具有相同线程属性组的一组线程。创建后，可以更改一些属性。有一些设置和读取这些属性值的函数。这些线程属性包括堆栈大小、优先权、规划策略、作用域、继承以及分离状态。对于作为一个分离线程来操作的线程可以设置为分离状态。表 3-3 列出 POSIX 线程属性对象的可设置属性以及简短描述。

表 3-3 POSIX 线程属性对象的可设置属性以及简短描述

属 性	描 述
contentionscope	决定线程将在哪一级上竞争资源：进程作用域或系统作用域。进程中的线程与该进程中的其他线程竞争资源。线程也可以与系统范围内全局分配的其他进程中的线程竞争资源。这个属性可以通过线程的局部作用域库进行规划，或通过全局作用域的操作系统进行规划
detachstate	允许其他线程等待指定线程的终止
stackaddr	设置指向线程堆栈的指针
stacksize	设置线程堆栈的大小
priority	设置规划类中线程的优先权
policy	设置规划策略（FIFO、轮询等等）
inheritsched	决定规划参数是否被继承或定义

线程信息和多处理器

线程属性对象可能包含处理器信息。在装有多多个处理器的系统中，可以将某线程分配给特定的处理器或处理器范围。在对称或非对称多处理器系统中，系统提供一个返回特定线程处理器亲合掩码的函数。线程亲合掩码是一个位数组，其中的每个位表示线程可以运行的处理器。因为只允许线程运行于它的进程允许运行的同一个处理器上，所以，线程亲合掩码是进程亲合掩码的一个适当子集。

3.8 线程创建

进程的主线程由操作系统自动创建。通过调用线程创建函数来创建后续或第二线程。当创建一个新线程时，通过程序的代码创建了一个新控制新流程。环境可以提供两种创建线程的方式：调用由线程库提供的函数，或者调用由操作系统提供的函数。由线程库提供的函数在进行一些内部处理后最终调用操作系统函数。

为了使用操作系统函数创建线程，系统必须知道线程函数的地址、传递给线程函数的参数、即将保存线程 ID 的变量、线程创建信息以及线程堆栈的大小。

当创建一个线程时，需要线程执行的函数的地址。函数在程序代码内定义，可能需要一个参数。如果需要多个参数，则可以声明一个结构。这个结构的地址是线程创建函数的参数之一。当创建线程时，系统返回线程 ID 并保存在一个专用参数中。表 3-4 列出了 OS/2、POSIX 和 Win32 环境中创建线程的函数，并对其中的参数作了解释。

表 3-4 在 OS/2、POSIX 和 Win32 环境中创建线程的函数以及参数的解释

环 境	函 数	参数与返回值	解 释
POSIX	int pthread_create(pthread_t *new_thread_ID, const pthread_attr_t *attr, void *(*start_func)(void*), void *arg)	*new_thread_ID	线程句柄
		*attr	指向属性对象的参数
		*start_func	指定线程函数地址的参数
		*arg	线程函数的参数
		返回值	返回一个 int; 0 表示线程成功创建, 非 0 值表示发生错误
OS/2	APIRET DosCreateThread (pptid Thread ID, ppThreadAddr, ulThreadArg, ulThreadFlags, ulStackSize)	pptidThreadID	保存创建线程的线程 ID 的双字地址
		ppThreadAddr	指定线程函数地址的参数
		ulThreadArg	线程函数的参数
		ulThreadFlags	设置此参数来决定创建线程是处于挂起状态, 还是立即执行。如果 0 位设置为 0, 则线程立即执行。如果 0 位设置为 1, 则创建线程处于挂起状态, 线程的创建者必须恢复它。如果位 1 设置为 0, 系统将使用缺省方法初始化线程的堆栈。如果位 1 设置为 1, 系统将预提交堆栈中的所有页
		ulStackSize	指定线程堆栈的大小

续表

环 境	函 数	参数与返回值	解 释
OS/2		返回值	返回一个 int; 0 表示线程成功创建, 非 0 值表示发生错误
Win32	HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpSa, DWORD ccStack, LPTHREAD_START_ROUTINE lpStartAddr, LPVOID lpvThreadParm, DWORD fdwCreate, LPDWORD lpIDThread)	lpSa	指向定义线程安全性属性的 SECURITY_ATTRIBUTES 数据结构的参数
		ccStack	指定线程堆栈大小的参数
		lpStartAddr	指定函数地址的参数
		lpvThreadParm	线程函数的参数
		fdwCreate	指定与线程创建有关的额外线程的参数。如果为 0, 则线程立即执行。如果值为 CREATE_SUSPENDED, 则线程处于挂起状态
		lpIDThread	保存线程 ID 的地址
		返回值	如果成功, 返回新线程的句柄; 如果失败, 返回 NULL

参数结构和线程 ID 不应该是局部声明变量。在新创建线程的生存期内, 调用线程可能会超出作用域。如果这种情况发生, 线程的堆栈将被释放, 而且销毁变量。

每个线程维护一个在进程内存中自动分配的堆栈。堆栈的大小必须足够大, 用于调用即将创建线程的所有函数、函数的参数以及由每个函数创建的局部变量。在线程终止时释放为堆栈分配的内存。

当创建一个新线程时, 同时创建了这个线程的上下文。线程的上下文在线程的执行期间得到保持。线程的上下文是一套寄存器、堆栈以及线程优先权。在寄存器组中有指令或程序计数器、标志、函数的返回地址以及堆栈指针。程序计数器指向下一条执行的指令。当线程首先执行时, 线程的状态由一个创建标志设置, 它可以为处于挂起状态创建的线程而设置。挂起线程直到调用恢复线程的函数时才会恢复。创建标志可以为立即执行的线程设置。之后, 线程将处于就绪状态。

线程的优先权对线程的上下文也重要。优先权让分派器知道哪一个线程将安排在下一个执行, 以及由于更高优先权线程可用了, 所以此线程应当被抢先。线程继承了创建它的线程的优先权。优先权 (priority) 包括优先类 (priority class) 和优先级 (priority level)。在某些系统中, 只有一个优先级。主线程由于被系统自动创建, 所以给它分配了被认为正常的优先类与优先级。创建线程后, 可以更改其优先类和优先级。

3.8.1 谁可以终止线程

线程在执行完毕后终止。当线程返回到创建它的函数时自动终止。线程的资源被释放。线程也可以显式调用一个线程终止函数。任何等待该线程的终止的线程都将收到终止线程 ID 或线程终止的信号。如果线程使用一个终止线程堆栈的指针, 指针将由于堆栈的销毁而变成无用。如果

终止线程为主线程，它将导致整个进程的终止。如果终止线程为进程的最后一个线程，是主线程之外进程的最后一个唯一存在的线程，则会导致进程的结束。

进程内的一个线程可以强迫另一个线程终止。线程不能强迫自己终止。强迫一个线程终止意味着该线程将在完成执行前终止。如果线程没有反应，或执行不正确，销毁此线程是必需的。因为进程内的所有线程共享内存空间，如果一个线程导致内存的崩溃，这将危及整个进程。系统可能被强迫终止整个进程。在此之前，可能需要减少线程导致的错误、销毁线程措施。只终止该线程而不终止整个进程，除非目标线程是主线程。所有由线程终止打开或创建的资源以及它的堆栈都被销毁。在某些系统中，堆栈不释放，因为其他线程可能正使用引用堆栈上数据的指针。此时，堆栈不会释放，直到整个进程被销毁时才释放。表 3-5 列出了 POSIX、OS/2 以及 Win32 环境中与线程终止有关的函数。

表 3-5 在 POSIX、OS/2 和 Win32 环境中与线程终止有关的函数

环 境	函 数	参数与返回值	解 释
POSIX	void pthread_exit(void *status)	*status	线程的退出状态
		返回值	无返回值
	int pthread_detach(pthread_t threadID)	threadID	被分离的线程句柄
		返回值	如果成功，返回 0。如果不成功，返回的非 0 值表示发生了错误
	int pthread_cancel(pthread_t thread)	thread	被取消的线程句柄
		返回值	返回一个 int; 0 表示线程的成功取消，非 0 值表示发生了错误
OS/2	APIRET DosExit(ULONG terminate_action, ULONG exit-code)	terminate_action	指定终止什么。如果值为 0，则终止当前线程
		exit-code	完成状态。
		返回值	返回一个 int; 0 表示线程的成功终止，非 0 值表示发生了错误
	APIRET DosKillThread(TID ThreadID)	ThreadID	将被销毁线程的线程标识符
		返回值	返回一个 int; 0 表示线程的成功终止，非 0 值表示发生了错误
Win32	BOOL TerminateThread (HANDLE hThread, DWORD dwExitCode)	hThread	终止线程的标识符
		dwExitCode	终止线程的退出码
		返回值	如果函数成功，返回值为 TRUE。如果函数失败，返回值为 FALSE
	VOID ExitThread(DWORD dwExitcode)	dwExitcode	调用线程的退出码
		返回值	无返回值

3.8.2 分离线程

分离进程为异步子进程，它不继承父进程的任何属性。它们用作后台进程，不需要键盘输入或屏幕输出。分离进程在终止时不返回到父进程。分离线程在这一点与分离进程相似。

当线程终止时，终止线程的 ID 和状态由系统保存。如果进程不需要知道线程何时终止，就可以创建为分离线程。当线程终止时，线程 ID 和完成状态不会保存，也不会释放由线程使用的任何资源。为了创建一个分离线程，在创建的同时设置分离标志。运行的线程也可以为分离的。

守护程序 (daemon) 是分离进程的一个例子。守护进程 (daemon thread) 大部分时间处于阻塞状态。它一直保持阻塞，直到系统中发生了某事件，它才变成激活。守护进程的一个例子是前面描述的打印线程，它一直等到将任务放到队列中。它将任务发送给打印机，然后一直等到有新任务放到队列中。daemon 可以由另一个分离线程终止。这样的线程也可以监视打印队列。如果在特定的时间段内没有任务放到队列中，它将销毁线程打印 daemon。

3.8.3 远程线程

在 Win32 环境中，一个进程中的线程可以创建另一个进程中的线程。这称做远程线程 (remote thread)。创建远程线程的过程与在同一进程中创建线程相似，除了需要一个目标进程的句柄之外。线程在目标进程的地址空间中执行。

3.9 线程堆栈

每个线程都有自己的堆栈。堆栈的大小在创建线程时被固定。线程堆栈在进程的地址空间的堆栈片断中分配。如果线程的创建者没有指定线程堆栈的大小，则由系统分配为缺省大小。堆栈的缺省大小与系统有关，它取决于线程的最大数量、进程地址空间的分配大小以及系统资源所使用的空间。堆栈必须足够大，适应于线程的任何函数调用、进程外部的任何代码（如库代码），以及局部变量存储。对于多线程进程，进程的地址空间必须足以容纳文本、数据片断以及它的所有线程堆栈。

如果线程堆栈相邻排列或与数据片断直接毗邻，如果发生堆栈溢出，可能导致数据崩溃，而且可能终止进程。线程试图访问超过堆栈大小的空间时，系统设计了几种不同的方式来阻止线程使进程内存崩溃。在线程堆栈外部指派一个危险区 (danger area)、危险页 (danger page) 或危险地带 (danger zone)，当受到侵入时，系统就得到通知。一旦侵入这些区域，就采取一些行动。发生的行为取决于系统以及它是如何分配堆栈上的提交页的。某些系统移动危险区，让堆栈增长。这可以通过警戒页 (guard page) 来实现。警戒页位于堆栈的顶部，其下方是堆栈的提交页。如果侵入警戒页，它就下移到另一个提交页，允许堆栈的增长。这一过程重复进行，直到耗尽堆栈。下面的页为未提交页。试图访问未提交页将导致堆栈侵入的发生。在另一些系统中，访问警戒页将导致内存信息转储 (core dump)，用它来分析判断发生的事情。在这些系统中，警戒页在堆栈的底部。如果发生数据崩溃，进程将终止。图 3-4 显示了在 POSIX、OS/2 和 Win32 环境里，一个多线程进程的线程堆栈布局。

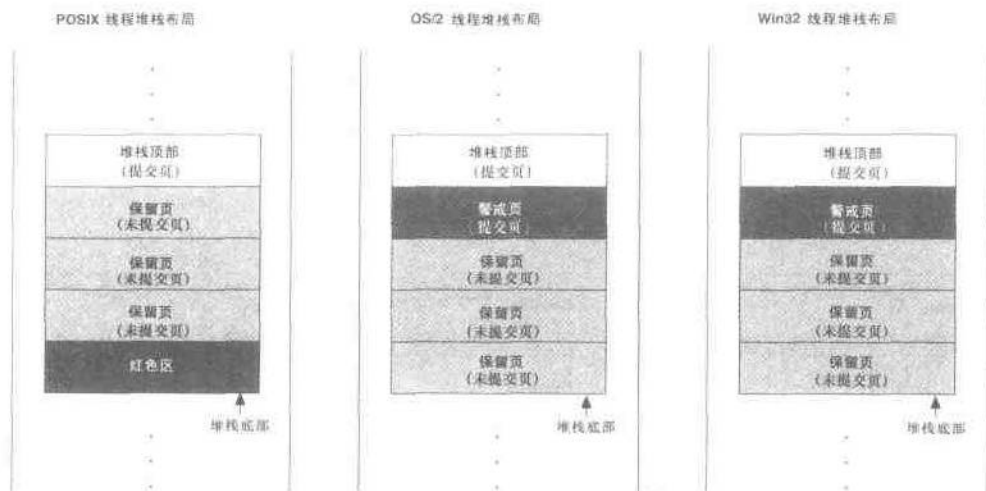


图 3-4 在 POSIX、OS/2 和 Win32 环境里多线程进程的线程堆栈布局

3.10 线程控制

尽管线程可以创建另一个线程，但它们与进程类似，也存在父-子关系。进程内的线程被看作同位体，因此对资源有同样的访问权限，相互间施加同等的控制。线程对父线程的资源、环境字符串以及看作全局的数据有相同的访问权限。线程能够访问由其他线程打开或创建的资源。如果一个线程创建了一个变量或打开了一个资源，该进程内的所有其他线程都可以访问它。这容易让线程间相互通信，而不必创建特殊的通信机制。线程既可以销毁也可以创建其他线程。线程可以销毁任何线程，包括主线程。之所以可以完成，是因为两个线程都位于同一个进程之中，而且当前线程具有目标线程的 ID。

3.10.1 临界区

临界区 (critical section) 是另一种类型的控制机制。线程的一个代码部分可以看作一个临界区。当线程进入一个临界区时，它访问与进程中所有其他线程共享的内存。当共享、可修改的数据正被进入临界区的线程使用时，它阻止其中任何线程的访问。所有试图使用这一数据的线程都将被阻塞。该线程退出后，才允许另一个线程来访问这些数据。一些系统的临界区实现时采取的措施更严格。它们阻塞整个进程中的所有线程，而不仅仅是阻塞试图使用数据的线程。临界区可以用于控制线程访问内存的权限。临界区可以用于读和写线程的问题。线程 A (进行写的进程) 不加区别地更新内存地址的数据。线程 B (进行读的进程) 读取这些数据，并应用在计算中。在线程 B 能读取这些数据前，线程 A 已经用一个新值覆盖了它。或者线程 B 第一次读取数据来进行计算，然后在线程 A 通过新值更新内存前再次读取数据，如图 3-5a 所示。这导致一些计算产生重复结果。两种线程都有临界区。线程 A 的临界区将阻止线程 B 在更新数据前

读取它。线程 B 的临界区将阻止线程 A 在读取数据前更新它。图 3-5b 演示了使用临界区解决读和写问题的方式。

其中有 3 个线程。因为线程 A、B 和 C 有不同的优先权，所以读取或更新资源的时间是不可预测的。

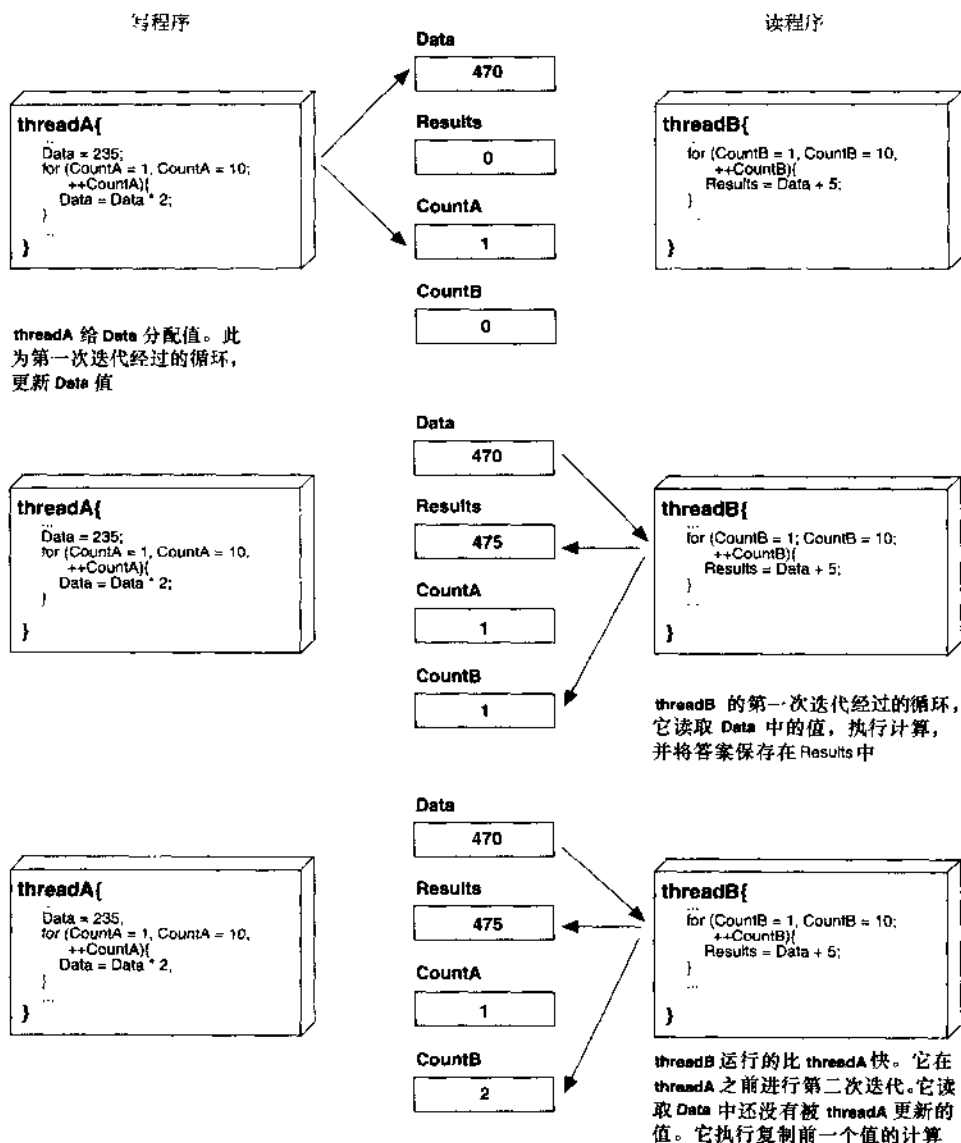


图 3-5a 读-写的难题。线程 A（进行写的进程）向内存地址写入值。线程 B（进行读的进程）第一次读取数据进行计算，然后在线程 A 用新值更新内存前再次读取数据

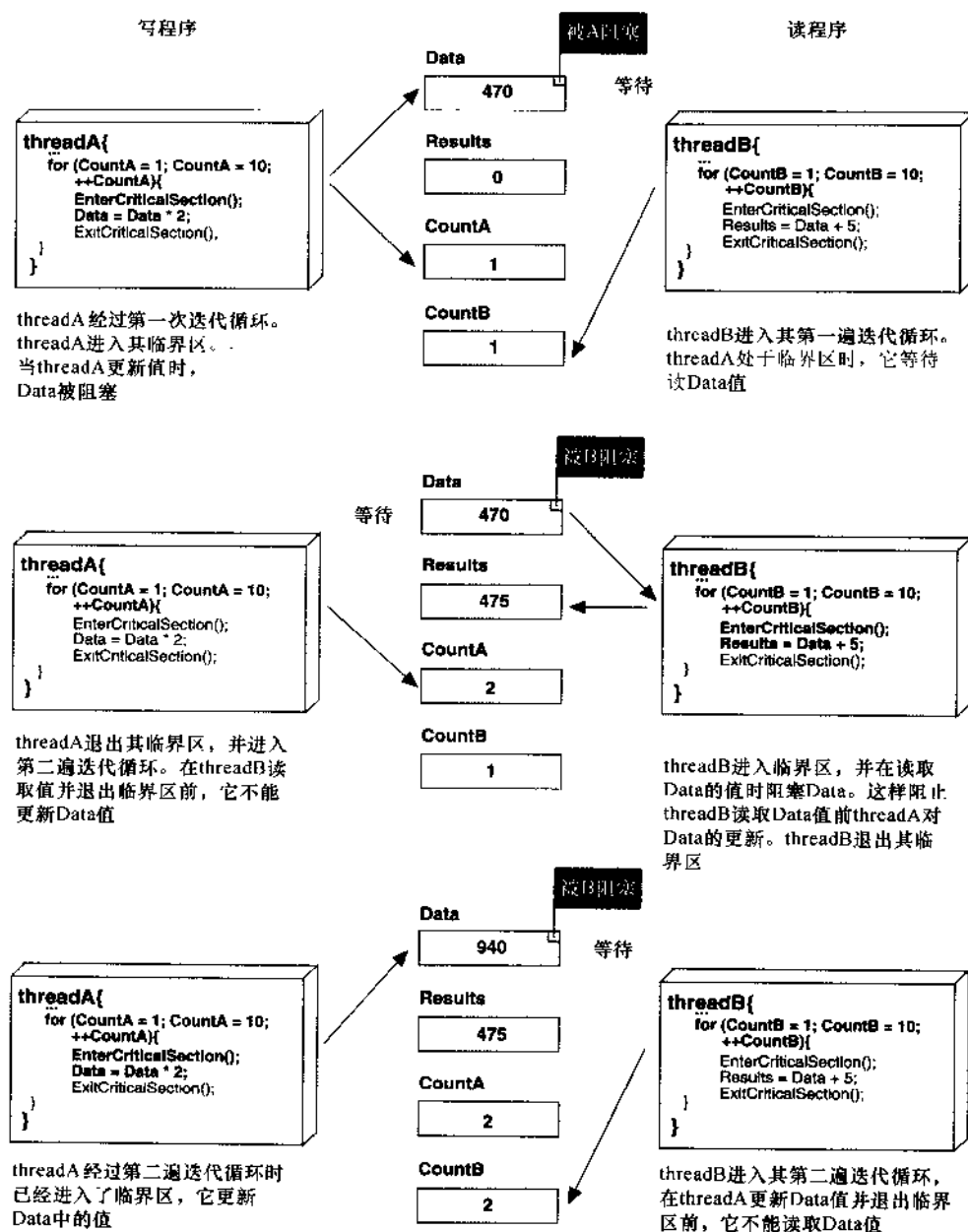


图 3-5b 演示临界区如何解决读-写难题

3.10.2 挂起和恢复线程

线程可以挂起进程内另一个线程的执行。挂起线程直到调用一个恢复它的函数时才会执行。

进程内的任何线程都可以恢复挂起线程，除了挂起线程自己。线程可以在指定时间段内挂起自身。

3.11 线程优先权

线程可以优先化。通过优先化线程，需要立即执行或对系统立即作出反应的任务就会分配需要的处理器时间。优先权规划用于决定哪一个线程使用处理器以及使用多长时间；CPU 时间在所有线程间不是平均分配。每个线程都有一个优先权，拥有最高优先权的线程在拥有较低优先权线程之前执行。在抢占式操作系统中，操作系统保持对处理器的控制权。如果较高优先权的线程可用、运行线程发出了一个 I/O 请求或被挂起，或者时间段用完了，正在执行的线程就会被抢占。因为每个进程都至少有一个线程，所以进程和线程优先权规划是相同的。

从就绪队列中，系统选择拥有最高优先权的线程来执行。就绪队列用一个有序列表来组织，其中每个元素位于一个优先级。列表中的每个优先级是一个具有相同优先级的线程队列。具有相同优先级的所有线程使用循环规划分配给处理器。操作系统允许队列中的第一个线程运行它的时间段。然后被抢占，让下一个线程使用处理器，直到队列中的所有线程都执行一遍，接下来，再次将第一个线程提交给处理器。图 3-6 显示了如何在就绪队列中放置线程，并且基于它们的优先权选择运行线程。

在 POSIX 环境中，同一优先级的线程可以使用 FIFO 或循环规划或其他策略来规划。规划策略还决定线程在队列中的放置地方。运用 FIFO（先进先出）规划策略，抢占线程放在同一优先级队列的头部。这意味着，该线程将要运行完成。当一个阻塞线程被唤醒时，它被放在同一优先级队列的末尾或尾部。当唤醒等待另一个线程的同步阻塞线程时，等待最长时间的最高优先权线程将首先取消阻塞。循环规划方案与规划线程的 FIFO 技术相似，只是运用循环规划时，如果运行进程时间段过期而且处理器交给了下一个线程，运行进程将自动被抢占。被抢占线程被放在队列的尾部。也允许使用其他一些自定义规划策略。例如，可以自定义 FIFO 规划策略，允许随机取消线程阻塞。

改变线程优先权

对于其他线程所依赖的线程，应当更改它的优先权加速它的运行。线程优先权不应该为了让特定线程得到更多的处理器时间而更改。这将影响系统的整体操作性能。高优先类线程比较低优先类接受更多处理器时间，因为它们执行更频繁一些。这看起来似乎对较低优先权的线程不公平。较高优先权的线程将主宰处理器，剥夺了其他线程有用的处理器时间。这就称做线程的饥饿（starvation）。使用动态优先权机制的系统将对这种情况和系统中的其他变化作出反应。分配优先权只持续短时间。为了给予低优先权线程执行时间、改进整个系统的操作性能，系统调整线程的优先权。

一、优先类和优先级

线程的优先权（priority）包括一个优先类（priority class）和一个优先级（priority level）。线程的优先类由它所属进程的优先类决定。新线程的优先级从创建它的线程继承。每个优先类都有一个级范围。在 OS/2 环境中，每个类有 32 个级，从 0 到 31。在 Win32 环境中，最高优先类有

16~31 个级，其他类有 0~15 个级。表 3-6 总结了 OS/2 和 Win32 环境中的类和级；两种系统都给原始线程分配一个缺省优先级。

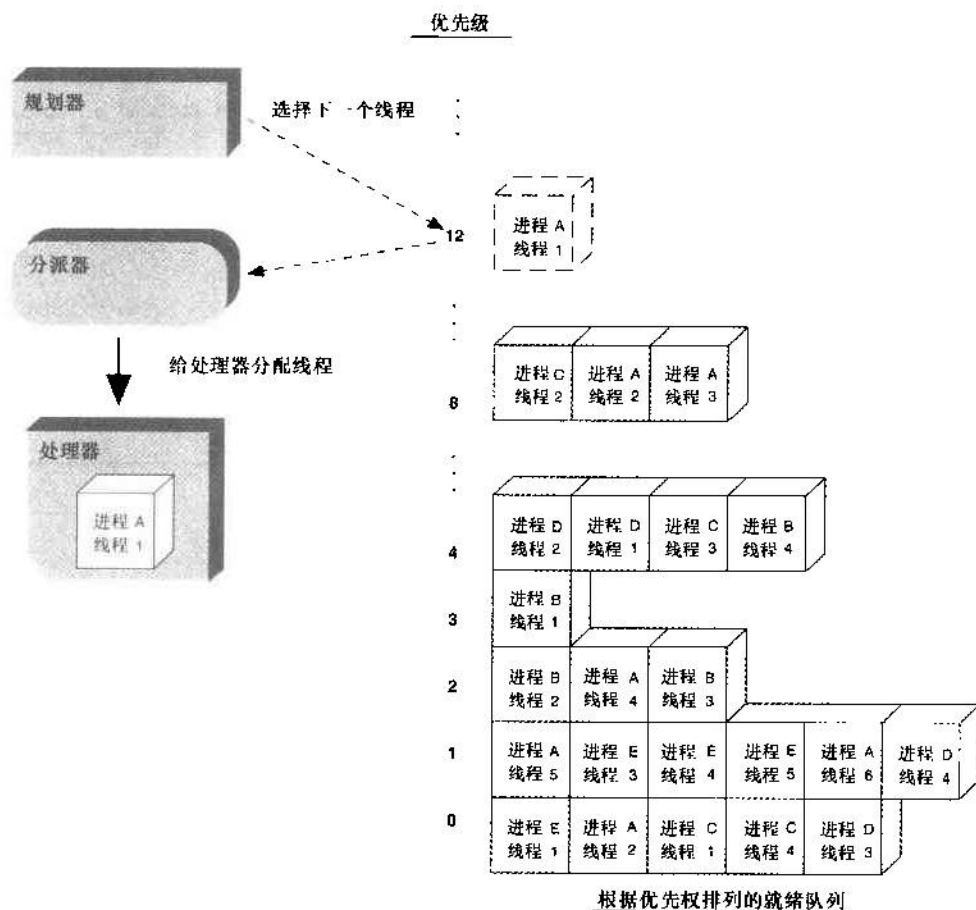


图 3-6 演示线程如何放在就绪队列中，并基于优先权选中线程来执行

表 3-6

OS/2 和 Win32 环境中的类和级

操作系统	类	级	描 述
OS/2	时间决定性 (Time critical)	0~31	时间决定性反应的最高优先权
	固定级别 (Fixed High)	0~31	针对以下任务：对于动态优先权提升不是太敏感；提供其他任务的服务；需要在常规优先权之前执行，但没有时间决定性
	常规	0~31	缺省优先权；大部分用户线路使用这个优先类
	空闲	0~31	最低优先权；仅在无其他优先权类任务时执行

续表

操作系统	类	级	描 述
Win32	实时	16~31	对于应当立即执行的短小任务的最高优先权；等同于系统函数
	普通	1~15	缺省优先权；大部分用户线路使用这个优先类
	空闲	1-15	最低优先权；仅在无其他优先权类任务时执行

线程的优先权称做基优先权 (base priority)。当线程的优先权改变时，它与它的基有关。线程可以改变它的优先权以及进程中其他线程的优先权。改变优先级是给基优先权添加一个值。如果级别提升了，则在基中添加一个正值。如果级别降低了，则添加一个负值。结果级限制于该类的级范围之内。表 3-7 列出了优先类与它的缺省优先级，以及 Win32 环境中更改线程基优先权的优先权标识。

表 3-7 优先类与它的缺省优先级 (Win32)

相对优先级 (Win32)	实 时	高	普 通	空 闲
初始级 (缺省)	24	13	7~9	4
空闲	16	1	1	1
最低 (-2)	22	11	5	2
低于普通 (-1)	23	12	6	3
普通 (0)	24	13	7	4
高于普通 (+1)	25	14	8	5
最高 (+2)	26	15	9	6
时间决定性	31	15	15	15

主线程可以更改它的任何线程的优先权。在 OS/2 环境中，如果子进程线程的优先权自从创建后就没有修改过，主线程可以更改它们的优先权。进程的优先类也可以变化。如果更改优先类，将同时改变进程中所有线程的优先权，同时更改子进程的线程还没有更改的缺省优先权。

二、提升线程优先权

正与用户交互的进程看作前台进程 (foreground process)，其他所有进程都是后台进程 (background process)。用户与进程交互的时候，前台进程应当得到充足的处理器时间，让系统对用户作出反应。如果系统中存在与前台进程具有相同优先类的进程，它们将平等竞争使用处理器。为了防止这种情况的发生，后台进程的优先权应该比前台进程的优先权设置得低一些。前台进程的线程先于后台进程的线路使用处理器。后台线程在没有前面线程准备执行时才使用处理器。后台线程甚至被可用的前台线程所抢占。所以，当后台进程变成前台进程时，该进程的所有线程都

将被推进（boost）或提升（raise）。这允许前台线程有能力对用户迅速作出反应。

线程的优先权有时也可能受到线程执行的干扰。当一个较低优先权线程锁定了一个资源，而这个资源正是一个较高优先权线程所需要的，这个时候就发生了这种情况。较低优先权线程不能释放锁定资源，因为另一个优先权较高的线程（但比需要此资源的线程优先权要低）已经阻塞了低优先权线程。若使用同步变量，如信号量的时候，以及低优先权线程进入一个临界区时，此时就发生这种情况。图 3-7 显示这种优先权倒置。

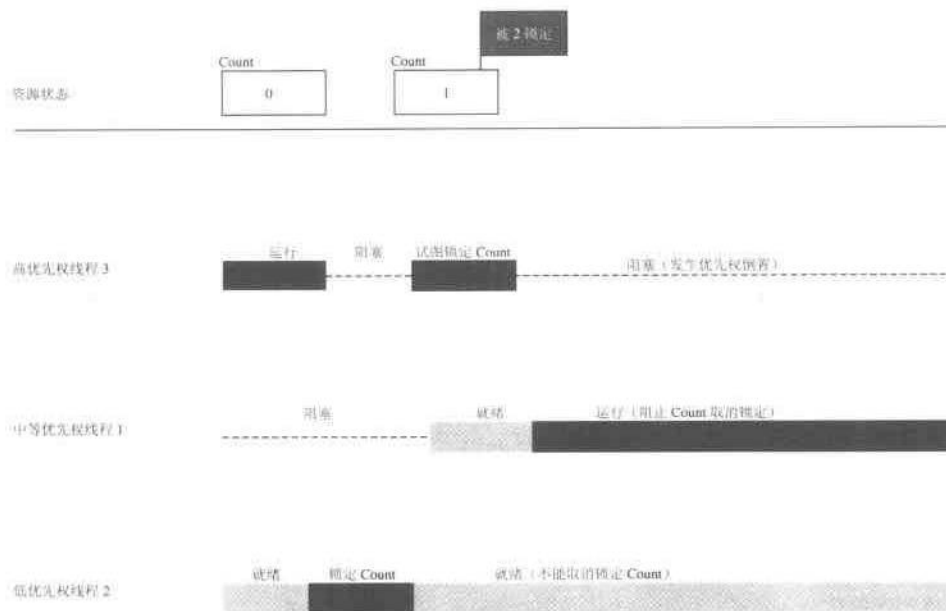


图 3-7 当一个低优先权线程锁定一个较高优先权线程所需要的资源时就发生优先权倒置。

较低优先权线程不能释放锁定资源，因为另一个较高优先权线程（但比需要该资源的线程优先权低）已经阻塞了较低优先权线程

线程 2 增加资源计数 Count，而线程 3 减少 Count。线程 3 在 3 个线程中优先权最高。线程 2 优先权最低。线程 3 首先执行。线程 3 和线程 1 阻塞允许线程 2 来执行。它锁定和增加 Count。图 3-7 显示了资源的状态。线程 2 增加和锁定 Count。在它能够取消锁定 Count 前，线程 3 和线程 1 变成非阻塞。因为线程 3 有较高的优先权，所以它被分配给处理器。线程 2 被抢占，但没有释放资源。线程 3 试图锁定 Count，但不能实现，因为线程 2 没有释放。线程 3 再次变成阻塞。线程 1 被分配给处理器。这将阻止线程 2 重新获得处理器并释放资源。线程 3 将继续处于阻塞状态，直到资源释放为止。这里就发生了优先权倒置。线程 1 阻塞时，线程 2 能够释放资源。

一些系统提供了处理这个问题的协议（protocol）。协议临时提升锁定资源的线程的优先权，防止被抢占，使得资源可以释放。这个优先权必须高于或等于能够锁定该资源的所有线程的优先权。锁定资源意味着，当使用该资源时，其他线程就不能访问它，直到使用它的线程完成。其他协议并不需要预先知道哪一个线程可以锁定资源。当资源被某个线程使用时，试图访问该资源的

线程被阻塞。使用资源的线程优先级被提升到高于阻塞线程，让这个线程使用和释放资源而不被抢占。图 3-8 演示了两种可以防止优先级倒置的方式。

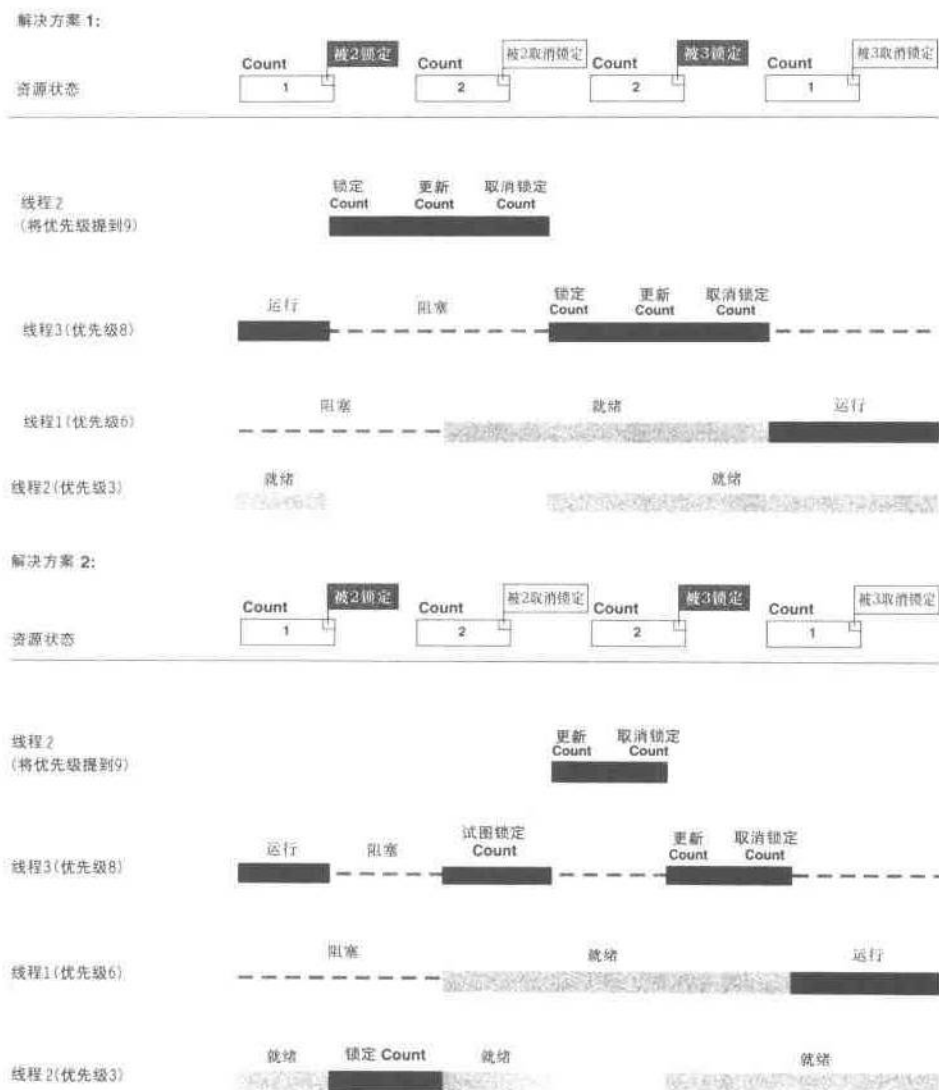


图 3-8 两种防止优先级倒置的方式。在解决方案 1 中，当线程 2 锁定 Count 时，它的优先级被提升到高于线程 3 的优先级。这就允许线程 2 锁定、更新和释放 Count，而不会被线程 3 或线程 1 抢占。在它释放 Count 后，线程 2 的优先级被减小到它的原始级别。在解决方案 2 中，直到另一个线程（即线程 3）试图锁定 Count 时，才提升线程 2 的优先级。当线程 3 试图锁定资源但失败时，线程 2 被抢占。它被阻塞。线程 2 的优先级被提升到高于线程 3 的优先级

在解决方案 1 中，当线程 2 锁定 Count 时，它的优先级被提升到高于线程 3 的优先级。这就

允许线程 2 锁定、更新和释放 Count，而不会被线程 3 或线程 1 抢占。在它释放 Count 后，线程 2 的优先权被减小到它的原始级别。然后，线程 3 可以锁定、更新和取消锁定资源。图示显示了 Count 被锁定、被线程 2 增加以及取消锁定的状态。线程 3 锁定、减小，然后取消锁定 Count。在解决方案 2 中，直到另一个线程（即线程 3）试图锁定 Count 时，才提升线程 2 的优先权。当线程 3 试图锁定资源但失败时，线程 2 被抢占。它被阻塞。线程 2 的优先权被提升到高于线程 3 的优先权。线程 2 然后可以增加和取消锁定 Count。再次降低它的优先权。现在资源就可用了，线程 3 解除阻塞、锁定 Count 并减小它的值，然后释放它。

三、优先权原则

当给进程和线程分配优先权时必须小心。用户倾向于给某个进程或特定线程分配最高的优先权，确保它完成运行。这将影响系统的整个操作性能。这些线程可能抢占网络通信，导致数据的丢失。包含此接口的线程可能受到严重的影响，导致键盘、鼠标和屏幕反应迟钝。优先权可能高于系统进程，阻止了它们对重要系统变化的反应。一般而言，大部分用户进程和线程归入普通或常规优先权之列。操作环境对于什么类型的线程应当分配高、中或低优先权提供了建议。表 3-8 列出了在 OS/2 和 Win32 环境中设置优先权的推荐值。

表 3-8 在 OS/2 和 Win32 环境中设置优先权的推荐值

类	级	推 荐
OS/2	时间决定性	20~31 机器人学/实时进程控制——针对控制设备的生产线。迟钝反应可能导致设备损害，人类伤害，甚至危及生命。最高优先权
		10~19 通信——对于防止数据丢失的通信或通信对话。次高级
		0~9 对于前台任务的抢占
	常规	26~31 通信——针对需要比后台任务优先权高的任务，增加与其他计算机传输和处理的交互
		0~25 缺省优先级为 0。用于将多个线程集作为不同优先级的任务，以优化它们的运行顺序
Win32	空闲	1~15 系统监视——针对那些周期性检查系统或应用程序，而不影响其他任务执行的任务
	普通	1~15 缺省优先级在 7~9 之间。针对管理前台和后台的任务。用户与某任务的交互使它成为前台任务，所有其他任务成为后台任务。系统自动将它的所有线程优先权提升 1
	高	1~15 仅在需要时使用。这些线程将与任务管理器（Task Manager）线程竞争，后者也处于同一优先权。任务管理器性能良好，而且时常需要 CPU
	实时	16~31 与硬件直接通信——极其高优先权。这种级别极少使用。大部分系统线程在较低优先权上执行。如果应用程序与硬件直接对话，它可以使用此优先权，或者针对那些非中断短期执行的线程

3.12 线程状态

线程是处理器上的可规划单元 (schedulable unit)。当规划一个进程来执行时, 使用处理器的是线程。来自系统中激活进程的所有线程都被放在就绪队列中, 根据它们的优先权进行混合和排序。当一个线程正在执行, 而另一个较高优先权线程变成可用时, 运行线程被抢占。新激活的线程可能与被抢占线程同属一个进程。如果这样, 那么就在两线程间发生上下文切换。如果新激活线程属于另一个进程, 则上下文切换发生在两个进程之间, 在其中进行线程上下文切换。

每个线程独立执行, 同时竞争处理器时间。它们与同一进程中以及其他进程中的线程进行竞争。每个线程都有自己的上下文和线程状态。第 2 章已经讲过, 线程具有相同的状态和转换过程。图 3-9 显示了线程及其转换的状态图。

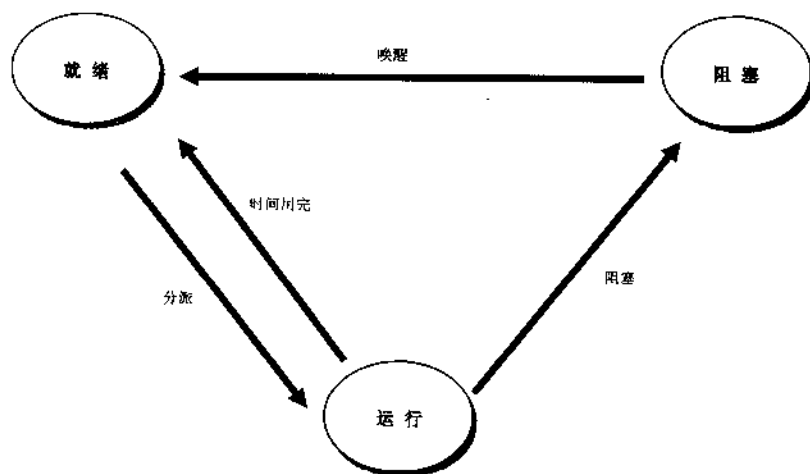


图 3-9 显示阻塞、就绪与运行状态以及相互间转换的状态图

回顾前面内容, 常见实现状态有 3 种: 就绪、运行 (激活) 以及阻塞。线程状态是线程在某个时间给定点所处的模式和条件。当线程准备执行时, 它处于就绪状态。直到它与其他准备执行的进程放入同一个队列中时, 才会执行。当选择某个线程来执行时, 线程就处于运行状态。如果线程等待某事件的发生, 它就处于阻塞状态。当线程从一种状态移到另一种状态时, 它就经历一种状态转换。因为某些事件的发生, 线程从一种状态转换到另一种状态。线程从就绪状态转换到运行状态, 这是因为系统选中此线程来运行。这种转换称做分派 (dispatch)。线程在处理器上运行短暂的时间段。时间段过期或线程发出一个 I/O 请求后或挂起后, 线程被抢先。如果时间段已经过期, 就将线程放回就绪队列中, 这时经历了时间耗尽或抢占转换。如果线程请求了 I/O, 则处于阻塞状态。第 2 章讨论了以上状态与其他一些状态以及它们之间的转换。

具有多线程的进程状态

进程有状态。单线程进程的状态与主线程的状态同义。如果主线程为挂起, 则进程也为挂起。

如果主线程恢复，进程也被恢复。如果主线程阻塞，则进程也阻塞。但当进程拥有多个线程，而且每个线程都有各自的状态时，进程处于什么状态呢？如果进程的所有线程阻塞，只有一个线程是激活的，则进程为激活。为了阻塞进程，进程的所有线程都必须处于阻塞状态。进程的所有线程为挂起，只有一个线程为激活，则进程仍然为激活。一个线程可以决定整个进程的状态。要让进程阻塞或挂起，进程的所有线程必须被阻塞或挂起。

3.13 线程与资源

与进程需求资源一样，线程也需求资源，如处理器、内存以及文件描述符。文件描述符被单独分配给每个进程，进程的线程可以访问这些描述符。在这种情况下，线程访问这些描述符时将与其他线程竞争。对于内存、处理器以及其他全局分配资源，线程都要与进程的其他线程以及其他进程的线程来竞争。所以，在进程内、系统内线程要竞争使用资源。一些操作系统允许用户设置单个线程的竞争域（contention scope）。

3.14 线程的实现模型：用户级线程

有3种线程实现。一种实现称做用户级线程（user-level thread），第二种实现称做核心级线程（kernel-level thread），第三种是以上两种实现的混合途径。图3-10对比了用户、核心以及混合线程实现。通过用户级实现，使用线程的程序链接到线程库。每个库函数调用都被翻译或映射属于进程一部分的运行时系统（runtime system）能够识别的调用。运行时系统实现线程管理。线程由库进行管理。这类线程包在操作系统的上层运行。对于操作系统，这些线程不可见。

操作系统的这类可执行可规划单元（schedulable unit of execution）为一个进程一线程（one thread per process）。不管库管理着多少线程，实际分配处理器时间的线程只有一个，即主线程。用户级线程具有多对一（many-to-one）线程关系。这种线程包开销低，因为线程不由操作系统管理。用户级线程的一个例子是C++任务库。NIH已实现了自己的用户级线程库。

3.14.1 核心级线程

核心级线程由操作系统管理。进程可以有多个线程，它们对于核心都是可见的。通过核心线程，每创建一个线程，都存在一个创建核心线程的系统调用。这是一种一对一（one-to-one）的线程关系。它们的开销比用户级线程大。对于用户级线程，这是一个重要的优点。不过，用户级线程的一个缺点是它们不能通过多个处理器使用系统。线程不能分配给不同的处理器。线程管理只发生于分配给该进程的单处理器上。通过核心线程，只要资源足够，可以给它自己的处理器分配线程。

3.14.2 混合线程

结合用户级和核心级线程的优点可以得到混合线程方式。在这种模式中，线程被看作一个用户级线程，但被映射到核心级实体。这种方式与核心级线程的不同之处是，只创建必需数量的核

心线程。可能许多线程被请求，但只能创建当前激活的线程。创建一个线程池（thread pool）从一个线程到另一个线程的模拟切换。线程库实际决定需要多少核心实体。POSIX.1c 使用混合线程实现。

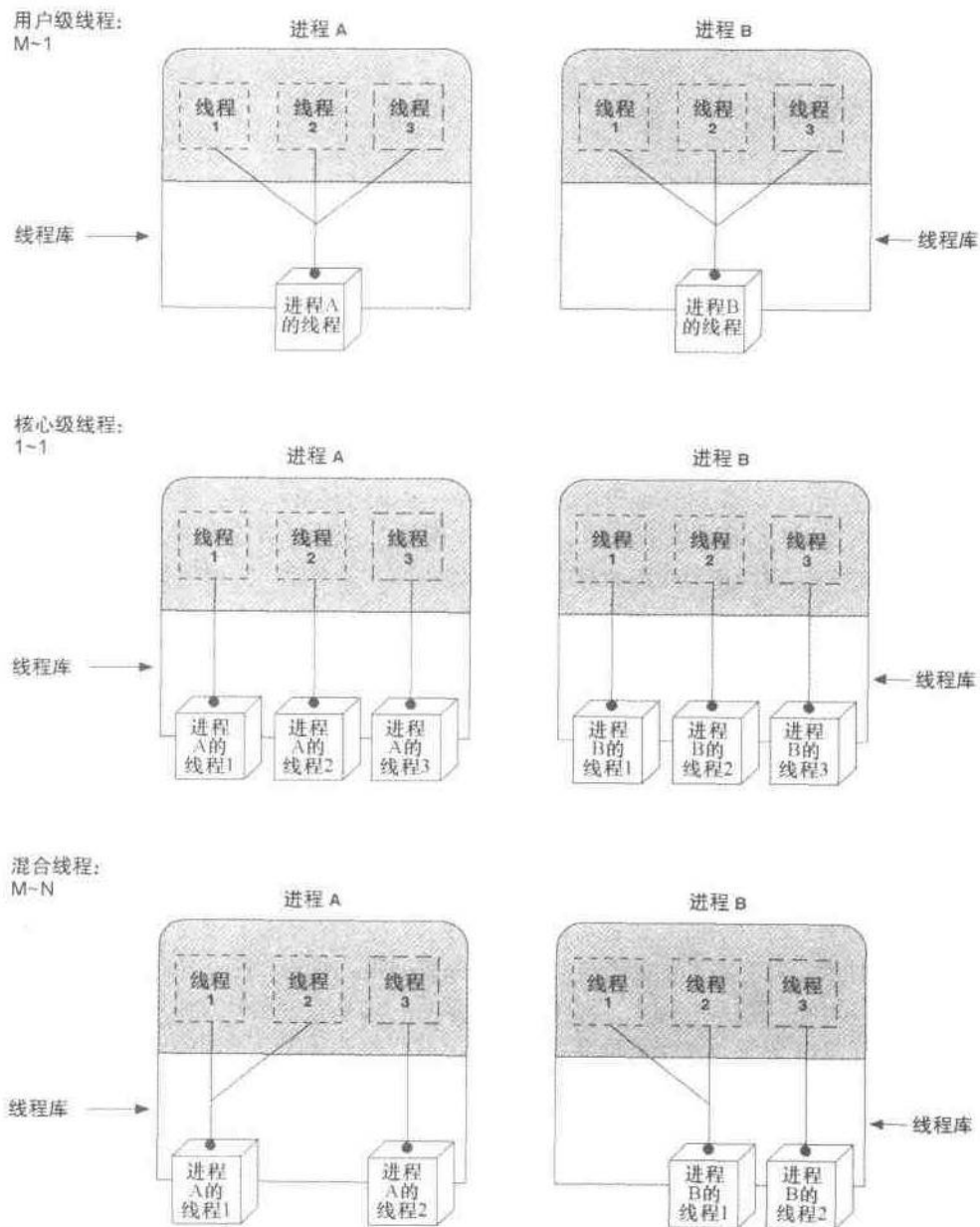


图 3-10 用户、核心和混合线程实现

多任务与多线程编程

...除非在解决问题时加以小心，否则，它们可能让解决方案完全丧失所有重要数字。

Cybernetics

——Norbert Wiener

多任务（multitask）同时执行多个进程，面多线程（multithread）允许单个进程同时执行多个任务。如果将一个进程分成多任务，而且每一个任务由一个线程执行，就称之为多线程。多任务和线程在许多方面相互关联。本章阐述在单处理器和多处理器系统中多任务与多线程期间发生的事情。

4.1 什么是多任务编程

当操作系统使用一个规划策略允许两个或更多进程并发共享一个 CPU 时，它称做多任务编程（multitasking），或多程序编程（multiprogramming）。在设计的时间段过期或某些事件发生前，一直执行某个进程。然后，操作系统切换到另一个进程。这种切换完成迅速，给人的错觉是，这些进程是同时执行的。而事实上，某时刻只能激活一个进程。这种进程间的切换在所有进程完成前一直进行。规划策略决定何时切换进程。规划策略可能由操作系统或其它进程强制执行。在某些情况下，由操作系统和进程两者负责强迫实施规划策略。多任务可以在 3 个级别上发生：

- 对话级；
- 进程级；
- 线程级。

图 4-1 显示了这些多任务和多线程级之间的关系。

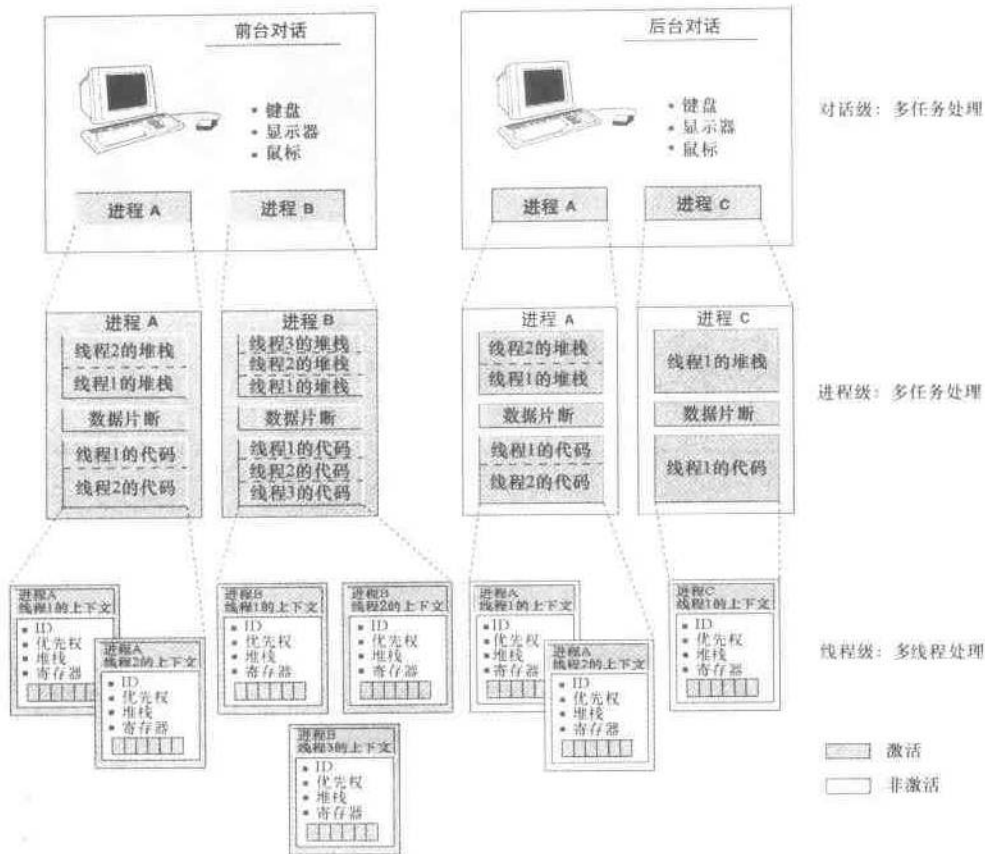


图 4-1 多任务和多线程编程对话和进程级间的关系

4.1.1 对话级多任务编程

对话间的多任务由用户控制。用户运行若干应用程序或对话时发生用户控制多任务 (user-controlled multitask)。对话创建一个虚拟计算机以及它的键盘、鼠标和屏幕。当执行一个应用程序或对话时，用户可以决定切换到另一个应用程序或对话。此时，执行两个应用程序或对话，一个在前台，另一个在后台执行。用户当前交互的应用程序或对话位于前台。用户可以在任何时候在它们之间切换，也可以决定终止哪一个应用程序或对话。因为由用户决定哪一个应用程序或对话位于后台或前台，以及终止哪一个应用程序或对话，所以称之为用户控制多任务。用户可以打开新应用程序或对话。

4.1.2 进程级多任务编程

在对话间，多个进程可以并发激活。进程可以相互合作来完成一个目标。例如，一个用户可能想要执行包含某命令集的所有 shell 脚本。进程的开始可能要搜索每个脚本寻找特定的命令列表。当发现包含此命令的脚本时，文件的名称就通过管道传给对话中的另一个进程。这个进程可

以在 `stdout` 显示包含此字符串的脚本。显示了脚本后，它可以通过管道将文件名传给即将解析和执行该命令的另一个进程。这些进程可以按序列一个一个地执行。第一个进程查找包含命令的所有脚本。第二个进程显示所有的脚本。第三个进程执行每个脚本。不过，可以在对话中使用多任务让所有三个进程并发共享 CPU。通过多任务，一个进程运行一段时间，然后另一个进程再运行一段时间。处理器改变每个进程的执行，直到所有的进程运行完成。对于以上描述的三个进程，用户不必等到找到所有的脚本后才显示和执行。在发现文件时，就可以显示和执行。当在一个对话中有多个进程激活时，这称做进程级多任务编程（process-level multitasking）。

4.1.3 多线程编程

所有进程由单独的任务组成，每个任务都有自己的控制流程。这些任务充当轻量级的进程，称做线程。进程的线程并发执行称做多线程编程（multithreading）。让我们详细看看这三级多任务是如何相互作用的。当用户使用鼠标或键盘上的键序列从一个对话移到另一个对话时，发生对话间的多任务。用户从一个对话进行切换，使该对话成为后台对话，同时作为前台对话激活另一个对话。当进行上下文切换（context switch）时，就发生进程间的多任务和多线程。当一个进程或线程被抢占或释放了处理器的控制，而且另一个进程或线程被分配给处理器时，就发生上下文切换。例如，在前一个脚本执行例子中，假定所有的 3 个进程正在一个优先权环境中执行，而且具有相同的优先权。搜索进程被分配给处理器。它执行了一段时间，然后被抢占或释放了处理器控制。如果搜索进程被阻塞、挂起或终止，显示进程被激活。在搜索进程和显示进程间发生上下文切换。显示进程被分配给处理器。当显示进程阻塞、挂起或终止时，解析进程激活。在显示进程和解析进程间发生上下文切换。这是进程间多任务的一个例子。

当进程的一个线程被抢占或释放了处理器控制，而且将另一个分配给处理器，此时就会发生线程上下文切换（thread context switch）。这种切换行为发生在进程的线程之间。解析器进程有多个线程。一个线程验证字符串，另一个线程解析字符串并执行命令。验证线程比其它线程优先权要高。执行命令的线程优先权最低，因为它只能在其它所有线程（除主线程外）已经完成后才能执行。验证线程首先分配给处理器。当它终止时，再将其它线程分配给处理器。当其它线程被分配给处理器并执行时，就发生线程上下文切换。然后抢占线程，或者其它事件发生，如得到处理器、线程终止、挂起或阻塞。这种活动在进程的所有线程运行完成前一直继续。这就是多线程的一个例子。

对话间的多任务是一个高级别的多任务，它受用户控制。进程间的多任务以及多线程在低级别上实现，并受到程序员的严重影响。程序员创建进程，并决定每个进程的线程数。程序员决定任务的优先权，以及什么时候挂起、什么时候终止。

多任务的目的是增加系统完成的工作量。多任务通过保持资源（处理器、I/O 等等）的繁忙而且试图防止它们在系统中处于非激活状态来使用资源。大部分任务不能不间断执行，因为它们等待 I/O 或另一个任务的终止，同时，其它任务可以使用资源。这有助于处理器更快地执行。

4.2 合作和抢占式多任务

多任务使用两种规划原则来实现：合作（cooperation）与抢占（preemption）。通过抢占多任

务，操作系统保持对处理器的控制。一旦任务提交给处理器后，它只执行短暂的时间段，然后，分配另一个任务给处理器。通过合作，任务分配给处理器后，它就不能被抢占。为了让另一个任务执行，运行任务必须自愿从处理器中删除自身。

4.2.1 合作多任务

合作多任务允许任务控制处理器。操作系统放弃对处理器的控制，而交给执行的任意任务。任务一旦拥有处理器后就可以选择运行任意长的时间。任务不会被系统操作抢占，除非它放弃了控制权。在这样的系统中任何任务都可能发生饥饿。合作多任务使用优先权规划。通常被看作后台任务的任務可以导致设备或任务间的通信，它们都需要来自系统的立即反应(均看作高优先权)，或者让它们等待。如果阻塞或挂起一个高优先权的任务，则低优先权任务被分配给处理器，它可以独占处理器，不允许高优先权任务有执行时间。然而，一旦低优先权任务分配给处理器后，它们可以在不被抢占的情况下运行到完成。

拥有处理器的任务可以在等待一个 I/O 请求完成时一直占据处理器。当 I/O 请求通过系统调用发出时，这个任务必须显式放弃处理器。系统依赖于任务间的合作来保持系统平稳运行。任务必须与所有的普通系统函数合作，否则，这些函数就会受到侵犯，因为任务正占据着处理器。通过合作实现多任务的方式：

- 设置它使用处理器的时间限制。
- 在代码中设置逻辑断点，在此处放弃处理器。
- 当发出 I/O 请求时放弃处理器。
- 当系统操作需要立即执行时放弃处理器。
- 放弃处理器允许交互任务使用处理器。
- 放弃处理器允许系统对中断作出反应。

合作多任务的优点是：实时、关键和通信任务可以运行到完成。下面列出了合作多任务的一些优点与缺点。

一、合作多任务的优点

- 对时间重要任务和通信任务不会被抢占。
- 程序员控制了系统的操作。
- 强迫程序员按防错方式设计任务，有目的地允许合作。
- 任务的执行和反应更具可预测性。
- 一种公平的规划方法：引入的高优先权任务不能取代已经运行的任务。
- 比抢占式规划开销低，抢占式规划因为频繁的上下文切换，所以需要在主存储器中保持多个任务。

二、合作多任务的缺点

- 允许任务独占处理器。
- 多任务只在任务得到处理器时才发生。由程序员决定是否发生多任务。
- 可能得不到可接受的反应时间。

- 程序员控制了系统的操作。
- 程序员在设计任务时，必须考虑系统的操作，而不是排它性地设计任务的功能性。
- 没有设计为合作的任务可以阻止正常合作任务的执行。
- 长时间运行任务将使短时间运行任务等待。
- 设计不良的任务可能导致整个系统停顿。

再看看前面讨论过的进程：搜索进程、显示进程以及执行 shell 命令的解析器。搜索和解析进程都是执行恰当的合作进程，但显示进程却不是。所有进程具有相同的优先权。相同优先权进程的规划将在本章的后面内容中讨论。这里，首先将显示进程分配给处理器。显示进程从一个管道中读取文件名，再显示文件的内容。因为它是首先执行的，搜索进程还没有分配给处理器，因此它没有在管道中放置文件名字。如果显示进程是一个恰当的合作进程，它将阻塞并释放处理器。但是因为它的设计不合理，所以一直控制着处理器。其它线程也就不能执行。图 4-2 显示了以上描述的情形。

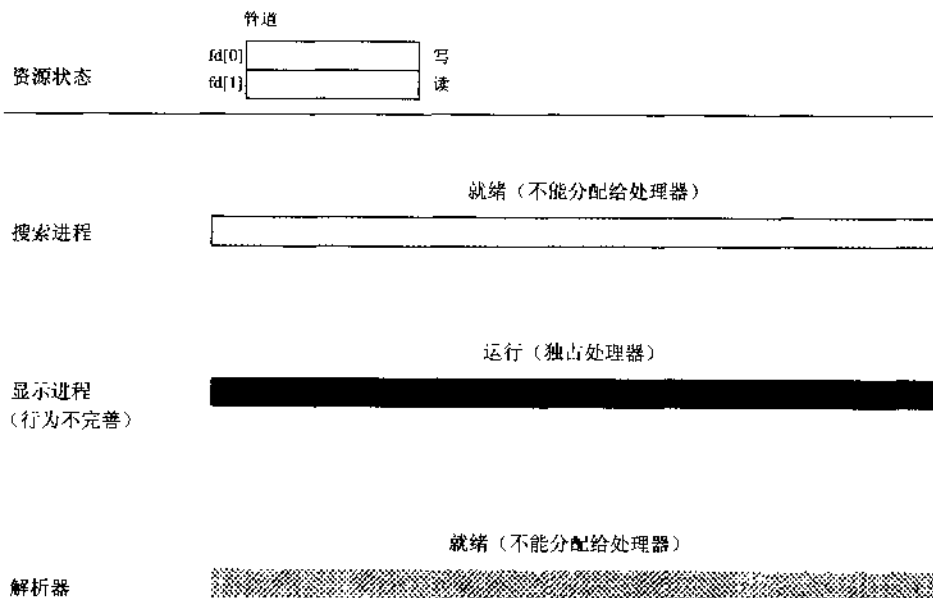


图 4-2 演示设计不合理显示进程以及搜索与解析进程的执行。显示进程独占处理器，阻止其它进程的执行

对于多线程进程，在进程的线程间进行合作是必需的。再看看第 3 章讨论过的 MCI 命令解析器。MCI 命令解析器由以下线程组成：验证字符串、提取噪音、符号化字符串以及执行。“符号化字符串”线程包括：获取命令符号、获取设备名字符号以及获取文件名字符号。这些线程必须合作允许进程的其它线程拥有处理器时间。

执行命令的线程优先权最低，因为它仅能在其它所有线程执行完成后才能执行（除主线程外）。首先将验证线程分配给处理器。“验证字符串”线程在这些线程中优先权最高，因为如果字符串无效，其它线程就没有一个对命令进行任何处理。“验证字符串”线程是执行进程的关键。它

可以不被抢占地执行。在抢占式系统中，由于它具有高的优先权，所以也可以执行到完成。线程可能被抢占，但可以重新获得处理器，因为它的优先权比其它线程更高。“验证字符串”线程判断字符串中有效后，然后终止执行。如果此线程占据着处理器，则其它线程没有一个能执行，进程被死锁。将处理器分配给下一个最高优先权线程，即提取噪音线程。这个线程从有效字符串中提取有效符号，去掉不必要的符号。这个线程也一直运行到完成，然后终止。接着将处理器分配给提取命令符号、设备名字符号以及文件名字符号的任何一个线程。这些线程优先权相等，可以按任何顺序执行。这些线程合作性非常强。它们执行短暂时间后，然后释放对处理器的控制，将其它某个线程分配给处理器。每个线程最终运行完成和终止。最后执行 MCI 命令。最后一个线程终止后，进程也结束。在这种情况下，所有的线程都运行至完成。大部分线程连续执行，而且没有利用多任务的优点。在多任务合作系统中，如果程序员在设计任务时不小心，系统可能降级为单任务。图 4-3 显示了这种情形。

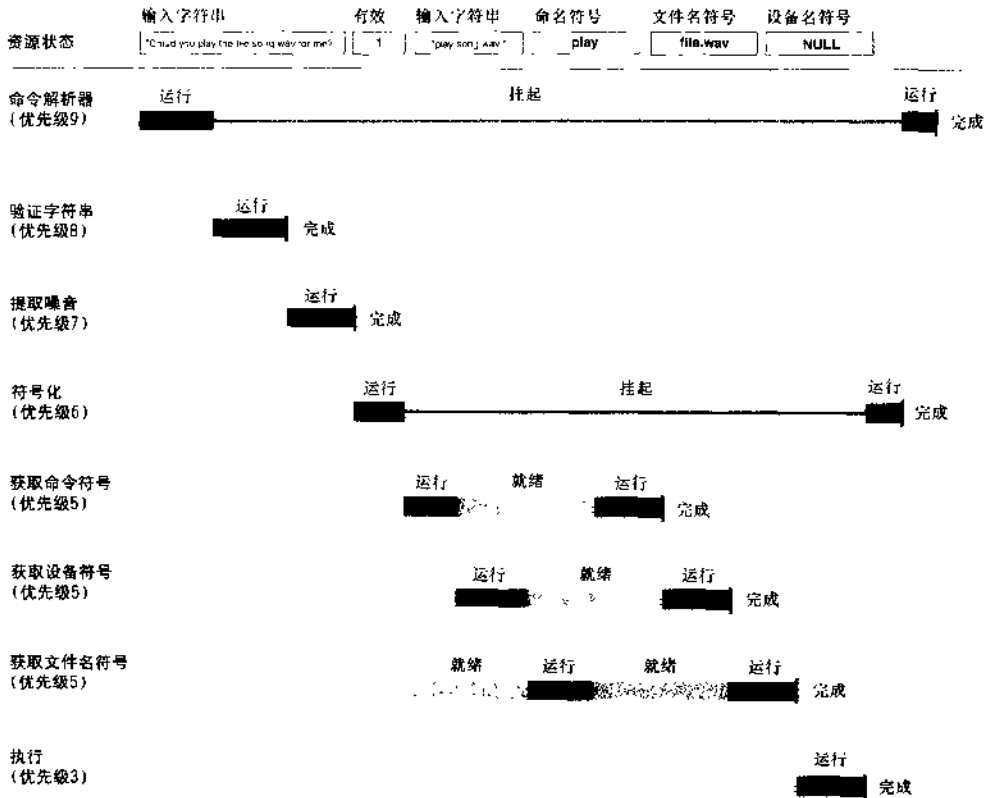


图 4-3 MCI 命令进程及其线程的执行

4.2.2 抢占式多任务

抢占式多任务是多任务的一种形式，在这种形式中，操作系统不会等到任务自愿将处理器交还系统，让它分配另一个任务给处理器。每个任务运行短暂的一段时间，称时间片断 (slice) 或

时间片 (quantum)。当时间片过期时, 操作系统抢占执行并分配另一个任务给处理器。这时就发生上下文切换。任务在时间段过期前, 可以自愿放弃处理器; 但如果仍然执行, 则该任务会被删除。抢占阻止任何任务独占处理器。当抢占一个任务时, 系统保存该任务的上下文。在抢占发生时, 操作系统使用上下文重新构建任务状态, 以便该任务再次提交给处理器时能够继续执行。下面描述了抢占式多任务的一些优点与缺点。

一、抢占式多任务的优点

- 不允许任何任务独占处理器。
- 设计不良的任务不阻止其它任务的执行, 而且不导致整个系统的停顿, 因为它们将被抢占。
- 操作系统保持对系统的控制。
- 设计任务时, 程序员不必考虑系统的操作, 而可以排它性地设计任务的功能性。
- 通过分配每个任务一段处理器时间来推进任务的执行。
- 由于上下文切换, 所以有可接受的反应时间。
- 新引入的高优先权可以抢占运行任务。
- 多任务由操作系统执行, 而不是由程序员执行。

二、抢占式多任务的缺点

- 时间敏感任务和通信可能被抢占。
- 比合作式多任务开销大, 因为频繁的上下文切换, 所以必须在主存储器中保持多个任务。

4.2.3 时间片的大小

一段时间称做时间片或时间片断。时间片的长度对于多任务和系统的操作非常重要。记住, 多任务的一个目的是增加系统完成的工作量。如果时间片太长, 它加长了一个任务的执行, 阻止了其它任务的合理执行时间。如果时间片太长, 一旦某个任务得到处理器, 它就执行到完成, 而不是多任务地执行, 系统操作性能就会降级到连续性执行。如果时间片太短, 开销变得更为重要。进程的开销是如此昂贵, 以致于要花费一定时间进行进程上下文切换。大部分时间用于进行这种切换, 花费在实际进程执行的时间并不多, 因此完成任务就很吃力。

时间片应当足够长, 让大部分交互式前台任务在时间片用完前有充足的运行时间发出 I/O 请求。发出 I/O 请求后, 该任务放弃处理器并被阻塞。这允许下一个任务使用处理器。时间片应当足够长, 让系统有充足的时间作出适当灵敏的反应, 将抢占开销降到最低, 并且让 I/O 资源使用效率最高。时间片的长度随系统的不同而不同。时间片的长度在不同的条件下也可能发生变化, 如沉重的系统负担 (大量任务要执行)。它还可能针对不同类型的任务而有所不同。例如, 一个经常使用处理器的任务, 通过让抢占尽可能地少, 它的运行时间可能比较充裕, 因此具有较长的时间片。但是, 不能忽略其它任务。I/O 范围的任务不需要长时间片。这种时间片应当让每个任务有充足的时间发出它的 I/O 请求。

更改时间片

一些系统允许更改时间片的长度。时间片有一个最小值和一个最大值, 以毫秒为单位。用户按这种方式定义时间后, 它就成为一个静态值。缺省时间片为动态。根据系统装载的不同, 时间

的大小也改变。对于允许这种变化的系统，在文件 config.sys 中定义时间片的大小。

4.3 多处理器下的多线程

“同时的多任务”这样的措词可以有不同的理解。它意味着执行多个任务，但一次只执行一个任务呢，还是意味着同时执行多个任务？对于单个处理器，使用抢占式或合作式规划，通过分配每个任务使用处理器时间来执行多任务。在抢占式多任务系统中，线程不断更改，直到完成任务。在合作式多任务系统中，线程必须在放弃处理器控制方面使用，以此允许其它线程的执行。其运作速度给人的错觉是，所有的线程都是同时执行的。单处理器上实现的多任务，任务并发执行。并发任务在同一时间内同时退出和执行。同时任务为并发，而且在同一时刻执行。异步执行也是并发，但能够在这些任务的不同执行时刻同时执行。在多处理器系统中执行的多个任务可以同时执行任务。多任务在多处理器上执行称做多处理器处理（multiprocessing）。表 4-1 列出了这些经常同义使用的术语，并指出了这些术语应用的环境。

表 4-1 经常同义使用的术语及其意义

术 语	意 义
并发（concurrent）	同时退出，而且在同一时间段内执行的任务。它们可能在同一时刻执行，也可能不在同一时刻执行。并发任务可以在单处理器或多处理器环境中执行。在单处理器处理环境中，两个任务同时退出，并通过使用上下文切换在同一时间段内执行
同时（simultaneous）	并发、且在同一时刻执行的任务。多处理器处理环境可以执行同步任务。其中一个任务在一个处理器上执行，另一个任务在另一个处理器上执行
异步（asynchronous）	任务可以为并发和同时，但不是必需的。它们在同一时刻开始。在不同的执行点，它们可能同时执行，而在另一时刻可能连续执行。一个任务可能在另一个任务终止前或终止后完成执行。异步任务可以在单处理器或多处理器处理环境中执行
多任务（multitask） 多任务处理（multitasking）	在单个处理器上执行多个任务，好像同时执行一样
多处理器处理（multiprocessing）	使用多个处理器，而且处理多个并发执行任务。每个处理器一个时刻执行一个任务
多程序（multiprogram） 多程序设计（multiprogramming）	两个或多个程序的并发执行。可以在单处理器或多处理器处理环境中执行多程序设计
多线程（multithread） 多线程处理（multithreading）	在单进程空间中，存在多线程执行。多线程处理可以在单处理器或多处理器处理环境中执行

多个任务可以在同一时刻执行，因为至少同时执行着两个线程。所有线程被分配给一个同时执行它们的单独处理器。如果线程设计合理，而且将线程间的资源竞争最小化，通过多处理器，它们可以比单处理器执行得更快。线程得到处理器的分配取决于多处理器处理系统为非对称组织，还是对称组织。

非对称和对称系统都处于一种紧密耦合（tightly coupled）环境中。紧密耦合环境意味着，处理器处于一种使用单一操作系统控制所有处理器的单一系统中。这些处理器有一个用于通信的共享内存区域。只要这里存在共享的内容，就有可能存在对它的竞争。通过在处理器间分布任务装载、使用阻塞或每个处理器的缓冲内存，可以将这种竞争降到最低限度。

紧密耦合环境与松散耦合环境相反。在松散耦合环境中，存在两个或更多的独立系统，它们各自有自己的存储器和操作系统。这种系统通过消息传递或远程过程建立通信链接，相互间可以访问对方的文件。在紧密耦合环境中，处理器访问相同的就绪队列。在松散环境中，独立系统保持有自己的单独就绪队列。单个进程的线程可以利用紧密耦合的多处理器。

4.3.1 非对称多处理器处理

在非对称组织的多处理器处理系统中，一个处理器执行一个设计好的任务。设计用一个处理器（譬如处理器 0）来执行输入和输出。其它处理器（譬如处理器 1 到处理器 n）执行计算密集型的任务。只要需要执行 I/O，就用处理器 0 来执行。处理器 0 只执行操作系统。用户任务不能在执行操作系统所在的同一个处理器上执行。因为单处理器运行着操作系统代码，这些代码不必重新进入。非对称多处理器处理用于非对称硬件上，例如一个处理器和一个协处理器（coprocessor）。

如果用户处理器（1 到 n）崩溃，不能执行任何操作，系统可以按一种有限的方式运行。如果处理器 0 崩溃，系统就不能执行任何操作系统功能，例如输入和输出。

从就绪队列中将线程分配给处理器。根据优先权，用户和系统线程在队列中混合、组织。系统将拥有最高优先权的线程分配给处理器。如果它是一个操作系统线程，则分配给处理器 0。如果是用户线程，则分配给处理器 1 到 n。如果没有操作系统线程执行，处理器 0 则处于空闲。如果其它处理器忙于执行用户线程，而且下一个用户线程等待执行，它也不能分配给处理器 0，即空闲的处理器。相反，如果处理器 1 到 n 可用，处理器 0 正执行，而且下一个分配给处理器的线程是一个操作系统线程，它也不能分配给处理器 1 到 n 中的任何一个。该处理器将一直处于空闲，线程也一直等到处理器 0 可用。非对称多处理器处理由于允许处理器处于空闲，所以减少了处理器的吞吐量。图 4-4 显示了多处理器处理的非对称概念。

4.3.2 对称多处理器处理

多处理器对称组织具备一个相互等同的处理器池（pool of processor）。处理器不是设计用于执行特殊类型的任务。任何处理器都可以执行系统或用户线程。任何处理器可以控制某个 I/O 设备或引用内存地址。操作系统线程可以在任何处理器上执行，而且从一个处理器漂移到另一个处理器上。所以，操作系统代码需要重新进入。某些系统有一个设计作为执行处理器的处理器。它负责系统数据和系统函数。对称多处理器处理是一种更稳固、可靠的环境。如果一个处理器崩溃，它就不再属于可用处理器之列。系统的执行级别稍微有所下降。

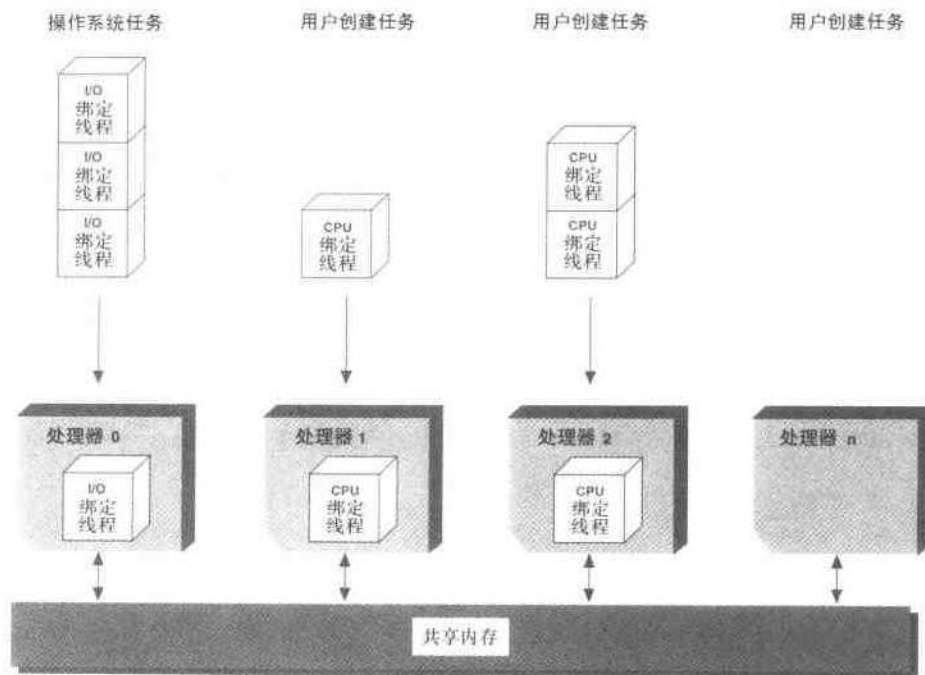


图 4-4 多处理器处理的非对称概念

所有处理器访问相同的就绪队列。最高优先级线程将被分配给任何可用的处理器。如果有一个可用处理器，而且存在准备执行的线程，则将某个线程分配给处理器。这将导致在系统工作负载与吞吐量增大之间取得更好的平衡。图 4-5 显示了多处理器处理的对称概念。

4.3.3 具有多处理器的多线程处理模型

创建具有同时执行多线程的进程强迫程序员按并行的方式来思考。考虑某时刻执行的行为要容易得多。程序员必须考虑在同一时刻所有执行的行为，以及它们之间相互影响的方式，或者一些全局结构。同时执行的任务可以相互独立发挥作用，或者在某个点，可能需要其它任务的同步或合作。需求同步化时，该任务可能不能继续执行，直到其中一个任务完成为止。需求合作可能意味着该任务从其它将完成工作的任务中获取一些数据。完成与其它任务独立发挥作用的同时 (simultaneous) 任务不需要同步 (synchronization)。事实上，它们可以不按特定的顺序来执行。它们可能不需要来自其它任务的数据，但它们可能给其它任务传递数据。进程可能需要结合这些任务类型来完成它的目标。

容易将完全独立的线程分配给它们自己的处理器来执行。其它任务可能需要一定程度的同步和合作。需要用一个范例为并发任务建立模型。模型将特征化如何将工作分解成线程，以及线程如何同步化。模型应当包含数据结构中的任何共享数据，使用某种类型的阻塞机制可以保护这些数据结构，使访问同步化而避免数据竞争。针对单处理器环境，这个模型应该取得一定的操作性能提高。

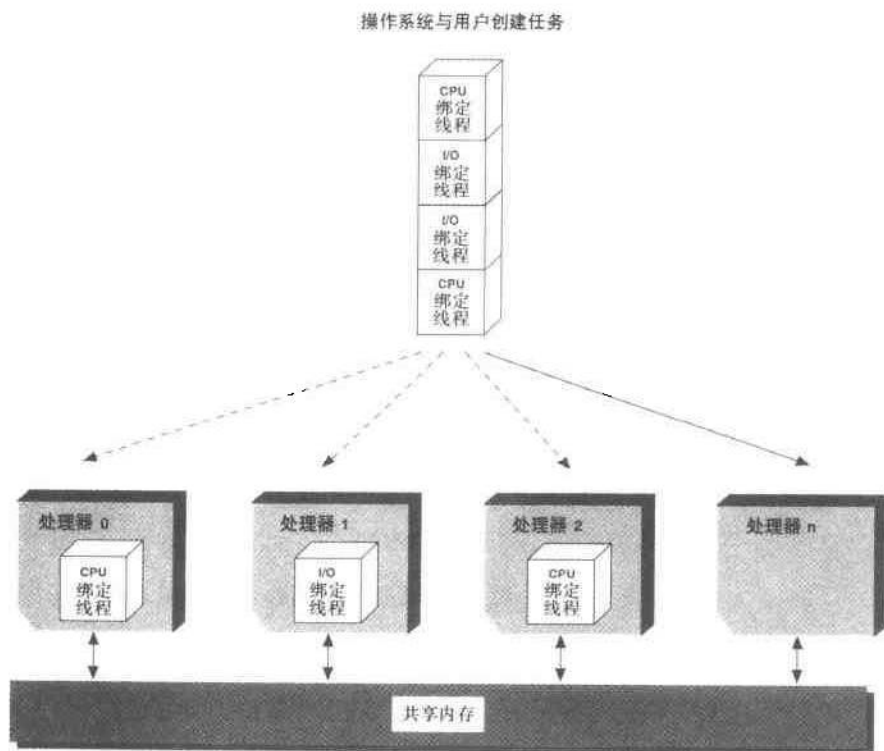


图 4-5 多处理器处理的对称概念

在多处理器处理环境中,一个用于多线程进程的模型例子是主-次线程模型(primary-secondary thread model)。在这个模型中,一个线程(指主线程)将任务分解,并分布到次线程中。这些次线程在同一时间调用。主线程一直等到所有的次线程都完成了它们的任务,然后才继续。主线程与次线程同步执行。主线程根据事先知道的工作调用线程。将工作大体平分为相等的部分。表 4-2 还列出了其它模型及其简短描述。

表 4-2

多线程处理模型及其简短描述

线程模型	描 述
主-次	主线程创建和启动次线程。为每个次线程分配一部分工作来执行。主线程预告知道工作量。在次线程间平均分配工作。主线程与它的次线程同步执行。它们执行任务,直到到达主线程与次线程间需要同步的点。在这个点,主线程访问工作判断次线程是否需要继续
管道线	工作分为几个执行阶段。这些阶段创建一个管道线或装配线,在每个阶段执行指定的任务部分。管道线中的每个阶段执行不同或相同任务。这些阶段并行执行。在管道线中为每个阶段分配一个线程

线程模型	描 述
工作堆	分配分成以队列形式存在于工作堆中的多个块。工作线程从堆中请求任务来执行。可以给工作堆添加工作线程。请求可以得到保证，直到工作堆为空。工作堆不同于主-次线程模型，主线程预告知道将要完成的工作量，然后创建相应数量的次线程。在工作堆模型中，预告不知道工作量。工作线程可能导致执行额外的任务

4.4 规划策略

非对称和对称组织决定在多处理器处理系统中如何将线程分配给处理器。规划策略决定什么时候分配处理器，以及分配哪一个线程给处理器。操作系统可能使用优先权规划，其中最高优先权线程被分配给处理器。当存在多个同一优先类和（或）优先级的线程时，使用另一个规划方案。

一些操作系统允许程序员所使用的规划策略，甚至允许有限制地自定义规划策略。规划策略可能不会得到预想的结果，对这些策略的自定义可能限制太严格。程序员必须设计和实现其中一种规划策略。

规划的发生级别有：

- 线程规划；
- 进程规划；
- 中间级规划。

线程是可规划执行单元。它们是实际分配给处理器的对象。进程规划决定激活哪一个进程。线程和进程在低级别上规划。低级别规划（low-level scheduling）由分派器执行。它在每秒操作多次，而且必须常驻在主存储器中。中间级别规划决定允许哪一个进程竞争处理器。竞争处理器的进程保存在就绪队列中。在这一级别，进程可以挂起，由于系统负载的变化也可能恢复。这个级别的规划是进入系统的进程与竞争处理器进程之间的一个缓冲器。

4.4.1 规划策略目标

规划策略的目的是决定哪一个进程或线程使用处理器，以及应当何时使用处理器。在优先权方案中，具有最高优先权的线程或进程被分配给处理器。决定哪一个线程或进程具有最高优先权是随机的。它取决于多个因素，例如系统的主要目的。如果系统要求密集的用户交互，那么系统对用户的快速反应至关重要。如果系统主要用于通信，那么所有通信进程或支持通信的进程应当具有一定程度的高优先权。如果系统是某制造过程或系统反馈的一部分，那么系统的变化至关重要。操作系统建议哪一个进程或线程应具有最高或最低优先权，但此类建议和原则可以被忽视。这些原则考虑了整个系统的操作性能。

优先权方案可以偏离系统。如果使用动态优先权，系统就可以短暂更改优先权，允许其它任务的执行。如果可能或切实可行的话，规划策略必须公平，平等看待所有任务。公平能阻止任务饥饿或无限延迟的发生。无限延迟可能导致死锁或系统的反应迟钝。反应迟钝看起来就像死锁一

样。某些系统为了避免无限延迟，实现优先权推进。优先权推进提升某进程或线程的优先权。

规划策略应当偏向于占有重要资源的任务。低优先权任务可能锁定资源，而这些资源将被其它具有较高优先权任务所使用，或被系统中的许多激活任务所使用。这可能会导致死锁，这种规划机制应当偏向于锁定这类资源的进程或线程，它因此可以释放资源。提高库提供的协议也可以解决这种问题。在实时（real-time）系统中，快速反应是重要的。资源必须在需要时可用。下面列出了系统规划策略的一些目标。

一、规划策略目标

- 最大化吞吐量。
- 强制优先权。
- 根据实际，尽量公平。
- 最小化开销。
- 避免无限延迟和饥饿。
- 偏向于占据了其它任务所需资源的任务。
- 在需要时，有充足的可用资源，让反应时间可接受。
- 不应当在重系统负荷下崩溃。

二、规划策略准则

- 任务使用处理器的时间仅足够产生 I/O 请求吗（I/O 绑定）？
- 任务在整个时间片内使用处理器吗（CPU 绑定）？
- 为了完成执行，任务需要多少处理器时间？与那些接受了更多时间的任务相比，应该偏向于没有接受更多处理器时间的任务。另一方面，这些任务可能接近完成。
- 是否任务频繁被高优先权任务抢占？在操作系统花费开销激活该任务后，它被抢占了吗？这些任务可能不太重视，但它们倾向于成为低优先权，因此这样的决策可能导致饥饿的发生。
- 该任务需要对系统的快速反应吗？实时任务需要快速反应。

4.4.2 规划策略准则

为了满足某些规划目标，必须考虑若干问题。例如，一些任务为 I/O 绑定（I/O bound），一些任务为 CPU 绑定（CPU bound）。CPU 绑定任务在整个时间片内使用处理器。I/O 绑定任务通常使用 CPU 的时间足以发出 I/O 请求。在抢占式系统中，它被阻塞到完成 I/O 请求为止。规划策略应当将 I/O 绑定任务分配给一个处理器，让它可以发出 I/O 请求并挂起等待请求的完成。当任务阻塞时，另一个任务可以分配给该处理器。这样可以推进任务的完成，有效使用系统的资源。在动态优先权方案中，I/O 绑定任务的优先权可以被提升到分配给处理器的级别。在非对称多处理器处理系统中，I/O 绑定和 CPU 绑定任务在单独设计的处理器上执行。

一些规划策略将偏向于那些还没有分配足够执行时间的任务。当在一个使用动态优先权的系统中发生饥饿时，低优先权任务被提升，让它能与其它独占 CPU 的高优先权任务一起完成。其它策略考虑任务离完成还有多远，或者考虑非常短暂的任务。接近完成的任务为侧重点，让它可以完成执行并退出系统。由于同样的原因，也偏向于短小的任务。这样的话，其它任务将不必长时

间等待分配给处理器，因为系统交给处理器完成的任务较少了。表 4-3 列出了部分规划策略及其简短描述，以及在实现策略时必须考虑的问题，包括某些规划策略的简短讨论。

表 4-3 部分规划策略及其简短描述

规划策略	非抢占式/抢占式	描 述
FIFO (先进先出)	非抢占式	根据放入就绪队列中的时间分派任务。一旦分配了处理器，它们将运行到完成。这是一个整体上平等的过程，但较长运行任务要运行到完成，让短任务一直等待。它不能保证良好的反应灵敏度，因为任务一旦激活后就不能被中断。它通常不用于主要规划方案，但用于具有相同优先权任务的优先性规划
SJF (最短任务优先)	非抢占式	首先运行预估运行时最少的任务。将该任务分配给处理器，然后退出系统。这将迅速减少等待任务的数量。需要预先知道每个任务的运行时。不能保证有灵敏反应时间
RR (轮询)	抢占式	根据 FIFO 分派任务。它们不运行到完成；它们运行一个时间片。如果任务没有完成，它将被抢占，并将处理器分配给下一个任务。被抢占任务放在就绪队列的后面。它可以保证可接受的反应时间
STR (最短剩余时间)	抢占式	具有最少完成预估运行时的任务下一个运行。具有较短运行时完成的新任务可以抢占激活任务。要求系统跟踪激活任务经过的时间。这将导致较高的开销
HRN (最高反应率其次)	非抢占式	每个任务的优先权由下面函数决定： $\text{优先权} = \frac{\text{等待时间} + \text{服务时间}}{\text{服务时间}}$ 一旦将任务分配给处理器，它就会运行到完成。偏向于较短任务以及长时间等待的较长任务

4.4.3 轮询和 FIFO 规划

在 POSIX 环境中，根据轮询 (round-robin) 或 FIFO，可以给处理器分配同一优先权的任务。在 FIFO 中，基于放置在就绪队列中的时间来分派任务。当某个任务被分配给处理器时，它将运行到完成。在抢占式环境中，当激活任务时间片过期后，该任务被抢占，但放在队列头而不是放在尾部，形成轮询。因为它放在队列的头，所以它将再次分配给处理器。这种情形一直延续到任务完成并退出系统为止。其它任务必须等到队列中位于它前面的任务完成执行。

FIFO 遵循一种队列中任务置换的策略。变成非阻塞的阻塞线程放在队列的末尾并赋予优先权。具有规划策略而且更改了优先权的激活线程放在新优先权队列的末尾。自愿放弃处理器的任

务放置在队列的末尾。FIFO 也决定了阻塞任务变成非阻塞的顺序。根据最高优先权任务阻塞最长时间，由于同步变量取消阻塞而阻塞任务。图 4-6 显示了 FIFO 规划策略。

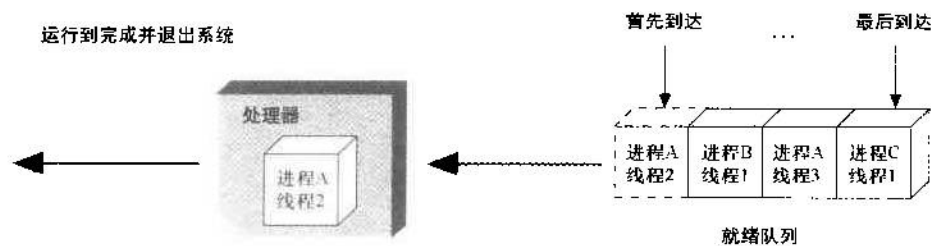


图 4-6 FIFO 规划策略

在 OS/2 和 Win32 系统中，使用轮询规划方案，将具有同一个优先类和（或）优先级的所有任务分配给处理器。在轮询规划方案中，根据带抢占的 FIFO（先进先出）将任务分配给处理器，它们只执行一个时间片，然后被抢占。接着将处理器交给队列中的下一个任务。抢占任务然后放在队列的后面。轮询方案保证队列中的每个任务都得到执行时间。所有任务都不会遭受饥饿。图 4-7 显示了轮询规划策略。

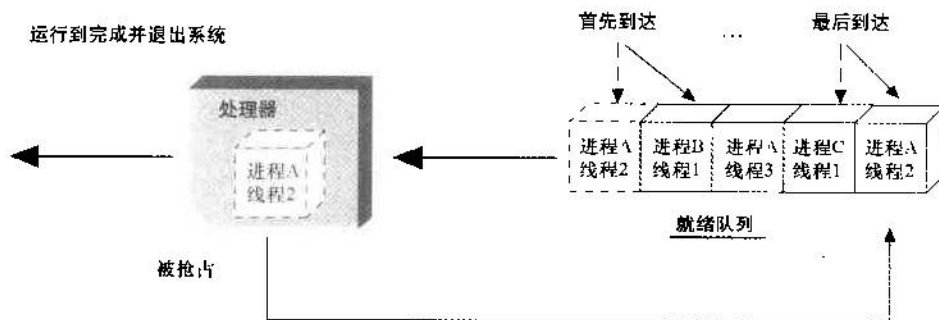


图 4-7 轮询规划策略

4.4.4 最短任务优先规划法

最短任务优先规划法（shortest-job-first scheduling）偏向于预估完成时间最短的任务。将短小任务分配给处理器，运行到完成并退出系统。在抢占式系统中，任务一直运行到时间片过期。它被抢占后，然后立即重新分配给处理器。如果新任务进入系统，它的预估完成时间比激活任务更短，则激活任务被新任务抢占。在短任务分配给处理器前，长时间运行的任务必须等待。首先执行短任务将迅速减少系统中任务的数量，缩短长时间运行任务的平均等待时间。

实现这一规划方案的难点在于，它需要预告知道每个任务的运行时间。此信息可能并不知道。在开发一个应用程序时，程序员必须估计任务运行到完成需要多长时间。如果估计不正确，规划系统可能必须抢占执行越过了估计时间的任务。同时，可能给早就执行完的任务分配了更多的执

行时间。图 4-8 显示了最短任务优先规划法。

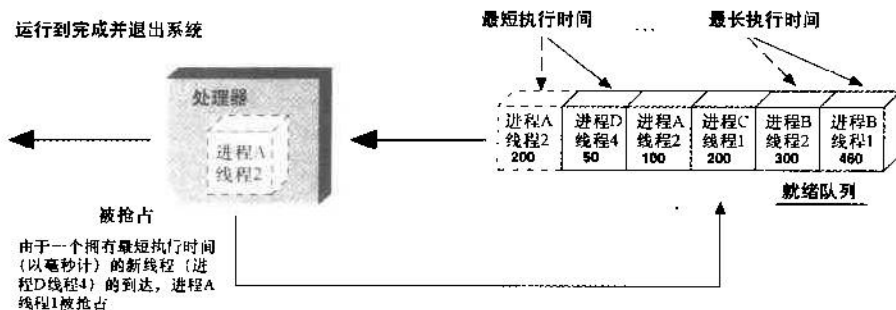


图 4-8 最短任务优先规划法。图中的数字代表每个进程的估计执行时间

4.4.5 最短剩余时间规划法

最短剩余时间规划法（shortest-remaining-time scheduling）策略偏向于具有最短剩余完成时间的任务。在这处规划方案中，一旦将它分配给处理器，该任务就运行到完成。如果系统中引入了剩余时间更短的新任务，则该任务被抢占。这种策略也需要预先知道每个任务的剩余运行时。它比最短任务优先规划法的开销要大。当激活任务被一个更短剩余执行时间的任务所抢占时，就发生上下文切换。

系统必须记录每个任务花去的服务时间。这给规划策略添加了额外的开销。图 4-9 示意了最短剩余时间规划法。

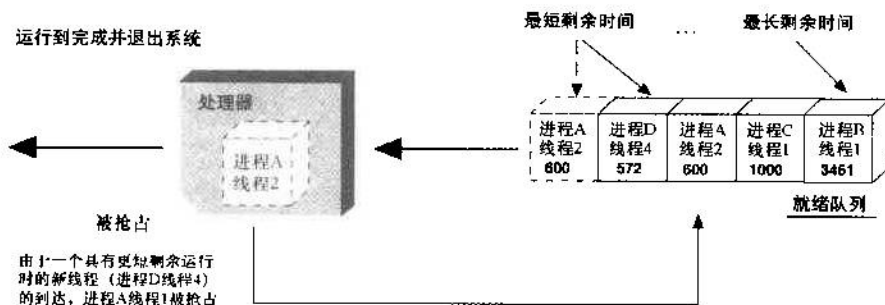


图 4-9 最短剩余时间规划法。图中的数字代表最短剩余执行时间

因为这种方案是抢占式的，所以几乎完成的激活任务将被一个已经进入系统，而且执行时间稍微短一些的新任务所抢占。如果任务是另一个进程，则发生进程上下文切换，这将使用更多的系统资源，而且增加了开销与时间。有必要使用一个值来表示该阈值（threshold value）。一旦激活任务的剩余时间小于此阈值，不管其它新任务的执行时间有多短，它都将执行到完成。

进程间和线程间通信

问题不是一种含糊，而是在规范之下的无助。

Language and Thought

——Noam Chomsky

许多程序和应用可以理解为一起工作达到某个共同目的的任务集。在不支持多线程或多任务的环境中，共同目标可能由一个任务在某时间来完成。每个任务在开始执行前等待前一个任务完成。例如，要求一个算术表达式的值：

$\sin(3+(7*(9/2)+11)-(4+3)/6)=2$

这个工作可以分解为几个任务。第一个任务必须从某些设备获取表达式并保存它。必须检查小括号是否匹配正确。还必须检查字符串是否包含正确的运算符和操作数。验证表达式后，必须求表达式的数字值。在一次只执行一个任务的支持环境中，表达式求值的每个方面必须按一种序列方式完成。与之相反，支持多线程的环境允许表达式求值的相关任务并发执行。例如，检查有效小括号、检查有效运算符以及检查有效操作数的任务可以并发完成。自然而然，表达式求值的任务必须等到以上任务完成后才能执行。

分配多线程来完成表达式求值的部分工作可能看起来相当简单，但它必须指出线程间通信和合作是不可少的。执行运算、操作数与小括号验证的线程只有等到将表达式读入内存后才开始。完成算术求值的线程要等到验证线程完成后才能开始执行。为了完成算术表达式求值这个共同目标，相关线程必须相互合作与通信。否则，线程很有可能产生一个不正确答案。

5.1 依赖关系

当线程或进程相互间需要通信或合作来完成一个共同目标时，它们就具有依赖关系（dependency relationship）。对于任意两个线程或进程，它们之间必定存在 4 种依赖关系。表 5-1

列出了这 4 种依赖关系。

表 5-1

线程间 4 种依赖关系

依赖关系	描 述
$A \rightarrow B$	线程 B 依赖于线程 A
$A \leftarrow B$	线程 A 依赖于线程 B
$A \leftrightarrow B$	线程 A 依赖于线程 B, 线程 B 依赖于线程 A
$A \oslash B$	线程 A 和线程 B 之间无依赖关系

如果存在线程 A 和线程 B, 线程 B 可能依赖于线程 A。在第二种情况下, 线程 A 可能依赖于线程 B。情况也可以是, 线程 A 依赖于线程 B, 同时线程 B 也依赖于线程 A。最后一种情况, 线程 A 和线程 B 间没有依赖性。也就是, 线程 A 和线程 B 可以相互独立。在第一种和第二种情况中, 依赖性为单向依赖性 (one-way dependency 或 unidirectional dependency)。在第三种情况中, 线程 A 和线程 B 相互依赖, 这是一种双向依赖性 (bidirectional dependency)。也就是说, 线程 A 和线程 B 之间的依赖性为双向的。任何两个线程间的依赖关系可以是以单向的、双向的, 也可以是无方向 (nondirectional), 即无依赖性。

5.1.1 通信依赖性

我们将着重讨论两类依赖性。当线程 A 需要来自线程 B 的数据进行操作时, 就发生第一类类型的依赖性。我们称之为通信依赖性 (communication dependency)。线程 B 必须在线程 A 继续执行前与线程 A 所获取的数据进行通信。例如, 假如在两个单独进程中执行的两个搜索程序间存在这种依赖性。第一个搜索程序命名为 findFiles()。这个程序搜索磁盘上的所有目录, 寻找具有特定扩展名 (txt、asc 或 doc) 的所有文件。当 findFiles() 找到符合搜索准则的某个文件时, 将这个文件保存在一个列表中。第二个搜索程序名叫 findWords()。这个程序搜索一个文件中的单词列表。当 findWords() 在文件中找到单词列表时, 它将文件放在一个列表中。我们可以构建一个派生两个并发执行了进程的程序。一个进程执行 findFiles(), 另一个进程执行 findWords()。如果磁盘、光盘或网络上包含正在搜索的关键词列表, 这两个进程于是可以一起工作, 来查找其中的所有文件。

在这种情形下使用多线程或进程比较方便, 因为磁盘、光盘和网络驱动的存储容量可能用 GB (gigabytes) 来计量, 有时用 1000 GB (terabytes) 计量。如果 findWords() 进程必须等到 findFiles() 进程完成搜索 GB 量的文件后才能执行, 那么 findWords() 以及其它依赖于 findWords() 的所有进程可以有效关闭, 不会产生不良影响。为什么? 当 findFiles() 在它的列表中保存一个文件时, findWords() 必须能访问这个列表, 读取其中的文件并开始搜索。因为文件可能也为 GB 单位级, 在 findFiles() 搜索时, findWords() 可能也在搜索。不过, findWords() 可能只搜索 findFiles() 返回的文件。在这里, findWords() 对 findFiles() 存在一个通信依赖性, 因为只有 findFiles() 生成了列表后, findWords() 才能执行自己的任务。在这两个进程间存在一种单向依赖关系。findWords()

依赖于 `findFiles()` 来完成列表，而 `findFiles()` 并不依赖于 `findWords()`。这种依赖关系只是单一方向。

5.1.2 合作依赖性

当线程 A 需要线程 B 拥有的资源，而且在线程 A 可以使用这些资源前，线程 B 必须释放它，此时就发生第二种依赖性。我称这种类型的依赖性为合作依赖性。如果有两个并发执行的进程，它们都试图同时访问相同的资源，那么，这些进程在成功执行前需要合作。我们可以使用前面例子中讨论过的两个进程 `findFiles()` 和 `findWords()` 来演示合作依赖性。如果我们声明一个全局变量 `Count` 来保存 `findWords()` 还没有搜索的列表中的文件数，当它向列表中添加文件时，让 `findFiles()` 增加 `Count`，同时，当 `findWords()` 搜索了列表中的一个文件后可以减小 `Count`。之后，我们必须提供一种方式让 `findFiles()` 和 `findWords()` 合作。如果访问变量 `Count` 不协调，`findWords()` 正试图减小 `Count` 时，`findFiles()` 可能试图增加 `Count`。因为 `findWords()` 减小 `Count` 时，`findFiles()` 不应当增加 `Count`，同时，当 `findFiles()` 增加 `Count` 时，`findWords()` 也不应当减小 `Count`。这两个进程有一种合作的依赖性。也就是说，它们必须合作才能成功完成共同的目标。`Count` 变量是某时刻只能由一个进程访问或占有的内存块。我们将在本章的后面讨论如何实现这种所有关系。

5.1.3 计数线程与进程依赖性

不管一个进程有多少线程，我们都可以通过比较进程内线程与其它每个线程的依赖性来理解整体线程关系。换句话说，如果进程有 3 个线程 A、B 和 C，我们可以检查 A 与 B、A 与 C 以及 B 与 C 来考察其依赖关系。一旦理解了进程内所有线程对的关系，该进程的整体线程结构就知道了。某进程内线程间的依赖关系数为 $P(n, k)$ ，其中 P 代表排列组合 (permutation)， n 代表进程中的线程数， k 代表依赖关系涉及的线程数。在这里， $k=2$ 。因此，对于以上进程，它有 3 个线程 A、B 和 C，线程间有 $P(3, 2)$ 种可能的依赖关系。也就是说，在这些线程间同时存在 6 种依赖关系。图 5-1 显示了线程 A、B 和 C 间可能的依赖关系图。

虽然在线程 A、B 和 C 之间只有 6 种协作关系，但这 6 种关系还有多个子集。对于两个线程 A 和 B，它们之间存在 4 种依赖关系。如图 5-2 所示。

图 5-2 中的关系可参见表 5-1。图 5-2 中的空关系 (null relationship) 与其它关系同样重要。当两个线程或进程间存在空关系时，就不需要通信和合作代码了。线程或进程相互独立执行。在这关系中，线程或进程不需要相互间的通信。空关系大大简化了应用程序中线程的使用。请注意，对应于两个线程，关系数为 4。如果我们将线程数量增大到 3，关系数为 2^4 。也就是说，在线程 A 与另外两个线程间存在 16 种依赖关系。图 5-3 显示了这 16 种关系的线程依赖性图。在头 6 个图中，线程 A 首先与线程 B 或线程 C 相关联。在下 10 个图中，线程 A 既与线程 B 关联，也与线程 C 关联。如果我们将线程数增大到 N ，则线程 A 与 N 个线程间的关系数为 N^4 。

线程依赖性图对于说明一套线程或进程间的依赖关系有用。依赖性图 (如图 5-3 所描述的) 可以用于与应用程序中相依赖的线程通信，而不需要通过源代码来搜索这种关系。这种信息可用于理解、保持以及测试多线程程序。

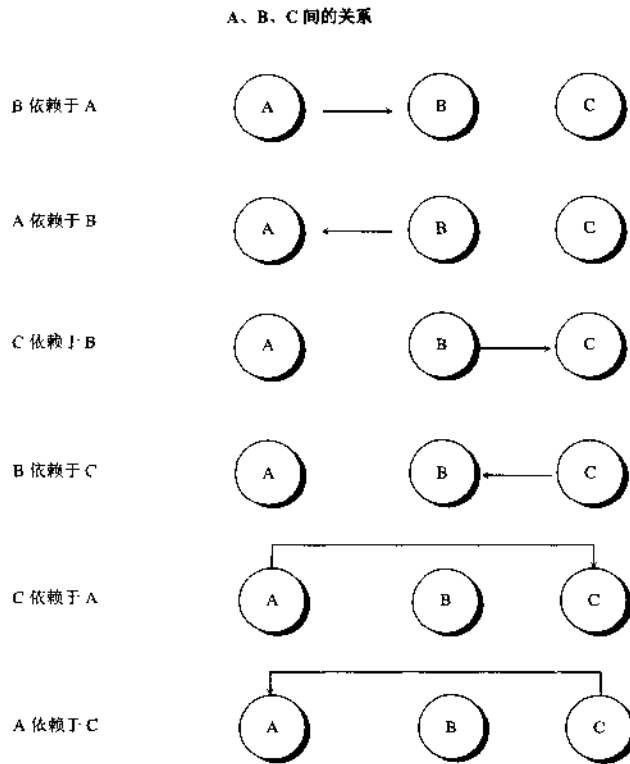


图 5-1 线程 A、B 和 C 间可能关系的依赖性图

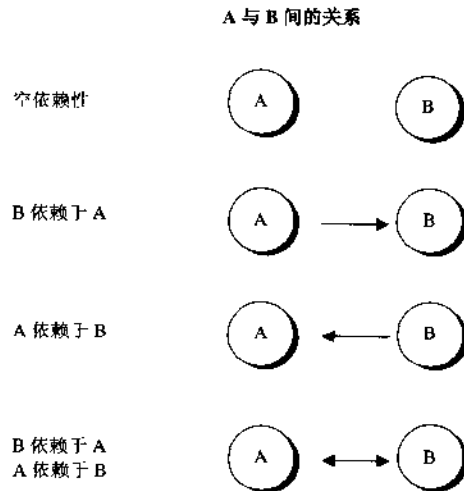


图 5-2 线程 A 与线程 B 间的 4 种依赖关系

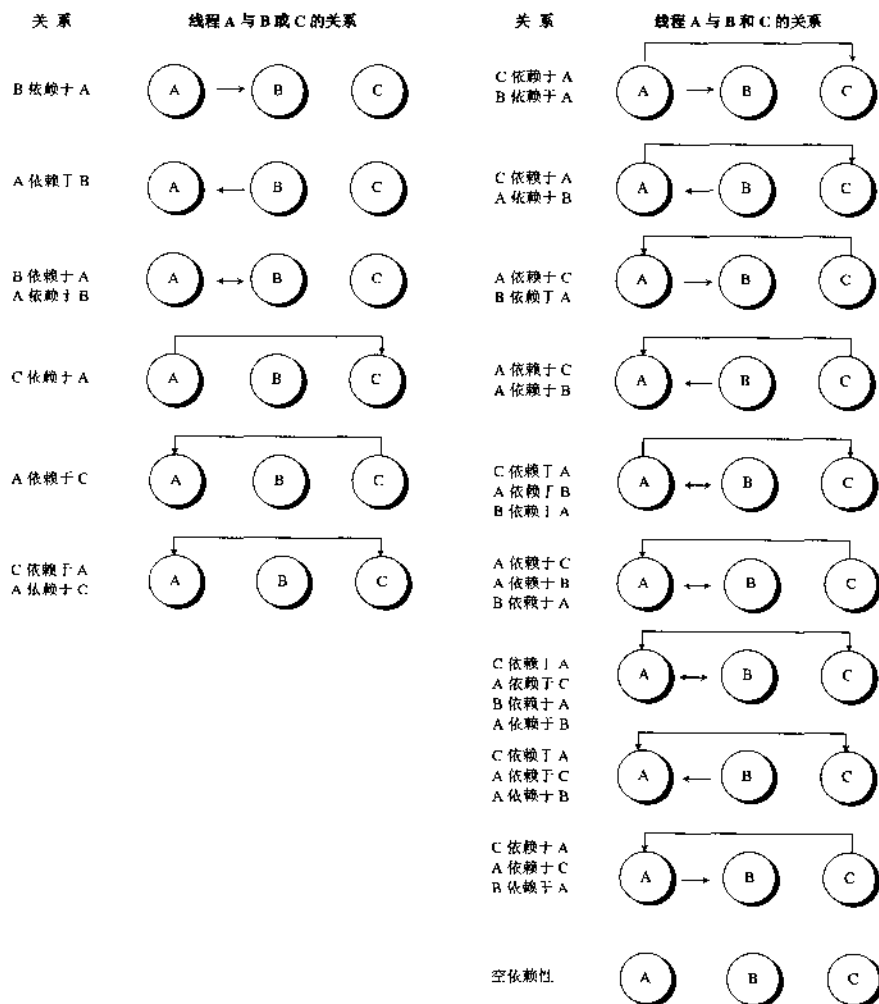


图 5-3 线程 A、线程 B 和线程 C 间的 16 种依赖性图。在头 6 个图中，线程 A 首先与线程 B 或线程 C 相关联。在下 10 个图中，线程 A 既与线程 B 关联，也与线程 C 关联

例如，`findWords()`和 `findFiles()`线程间的依赖关系可以用图 5-3 中的图 1 表达。在图 1 中，`findFiles()`用线程 A 表示，`findWords()`用线程 B 表示。该图显示线程 B 依赖于线程 A 提供信息或合作，但线程 A 不依赖于线程 B。依赖关系可用于任何数量的线程或进程。不过，对于数量多于 4 的线程或进程，此图将变得难以表示。例如，如果我们只在图 5-3 的图中再添加一个线程，则需要 81 个这样的图示。如果将线程数增加到 5 个，显示线程 A 与另外 4 个线程间依赖性所需的图示数量为 256 个。正如你所看到的，依赖性图仅对于少量线程或进程有用。依赖性矩阵 (dependency matrix) 可用于说明较大数量线程或进程间的关系。

图 5-4 使用线程依赖性矩阵显示了 4 个线程间的依赖关系。在依赖性矩阵中，依赖性按行

填充。

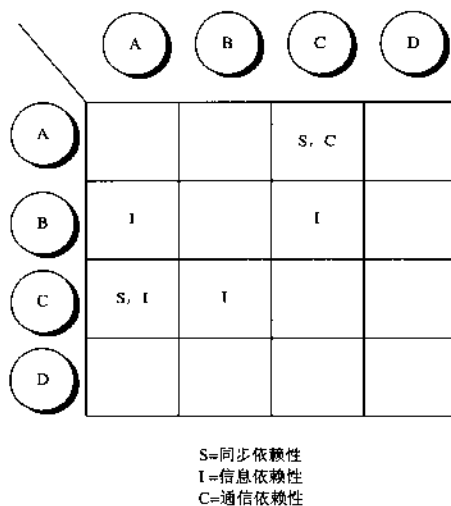


图 5-4 线程 A、B、C 和 D 之间的依赖性矩阵。在依赖性矩阵中，依赖性按行填充。“S”表示存在一种同步依赖性。“I”表示存在一种信息依赖性。“C”表示存在一种通信依赖性

在图 5-4 的矩阵中，A 行显示线程 A 对于线程 C 有同步和通信依赖性。B 行显示线程 B 对于线程 A 和线程 C 有信息依赖性。C 行显示线程 C 与线程 A 有同步和信息依赖性，对线程 B 有信息依赖性。D 行显示线程 D 并不依赖于任何其它线程。请注意，矩阵可以显示依赖性 is 单向的、双向的，还是空关系。例如，图 5-4 中的矩阵显示线程 B 依赖于线程 A，但线程 A 并不依赖于线程 B。

5.2 进程间和线程间通信

程序员必须让拥有依赖关系的进程集或线程集协调，这样才能达到进程或线程的共同目标。可以使用两种技术来达到协调。第一种技术在具有通信依赖关系的两个进程间传递信息。这种技术称做进程间或线程间通信（interprocess or interthread communication）。第二种技术是同步，当线程或进程相互间具有合作依赖性时使用。这两种类型的依赖关系可以同时存在。这意味着，一套线程之间可能既具有通信依赖性，也具有合作依赖性。

5.2.1 什么是进程间通信

一般而言，进程有单独的地址空间。这意味着如果我们有两个进程（进程 A 和进程 B），那么，在进程 A 中声明的数据对于进程 B 不可用。而且，进程 B 看不到在进程 A 中发生的事件，反之亦然。如果进程 A 和 B 一起工作来完成某个任务，必须有一个在两个进程间通信信息和事件的方法。在第 2 章描述了基本的进程组件。注意进程有一个文本、数据以及堆栈片断。进程可能

也有从自由存储空间中分配的其它内存。进程所占有的数据一般位于数据片断、堆栈片断或进程的动态分配内存中。数据对于其它进程来说是受保护的。为了让一个进程访问另一个进程的数据，必须最终使用操作系统调用。与之类似，为了让一个进程知道另一个进程中文本片断中发生的事，必须在进程间建立一种通信方式。这也需要来自操作系统 API 的帮助。当进程将数据发送到另一个进程时，称做 IPC (interprocess communication, 进程间通信)。表 5-2 列出了不同类型的进程间通信及其描述。

表 5-2 不同类型的进程间通信及其描述

进程间通信	描 述
环境变量/文件描述符	子进程接收父进程环境数据的拷贝以及所有文件描述符。父进程可以在它的数据片断或环境中设置一定的变量，同时子进程将接受这些值。父进程可以打开文件，同时推进读/写指针的位置，而且子进程使用相同的偏移访问该文件
命令行参数	在调用 exec 或派生函数期间，命令行参数可以传递给子进程
管道	用于相关和无关进程间的通信，而且形成两个进程间的一个通信通道。通常使用文件读和写程序访问
共享内存	两个进程之外的内存块，两个进程均可以访问它
DDE (动态数据交换, dynamic data exchange)	使用客户机/服务器模型。服务器对客户的数据或动作请求作出反应

5.2.2 进程间通信类型

当一个进程派生出其它进程时，我们说派生的进程是相关联的。派生进程为子进程，派生它们的进程称为父进程。关联进程使用一套技术来执行进程间通信，而不相关联的进程通常使用另一套技术来通信。我们首先讨论关联进程使用的进程间通信类型。

一、环境变量、文件描述符

当创建一个子进程时，它接受了父进程许多资源的拷贝。例如，当在 UNIX 环境中作为子进程创建 fork() 调用时，子进程接受了父进程的文本、堆栈以及数据片断的拷贝。子进程也接受了父进程的环境数据以及所有文件描述符的拷贝。子进程从父进程继承资源的过程创造了进程间通信的一个机会。父进程可以在它的数据片断或环境中设置一定的变量，然后执行 fork()。子进程于是接受这些值。同样，父进程也可以打开一个文件，推进到文件内的期望位置，然后执行 fork()。子进程接着就可以在父进程离开读/写指针的准确位置访问该文件。

这类通信的缺陷在于它是单向的、一次性的通信。也就是说，除了文件描述符外，如果子进程继承了任何其他数据，也仅仅是父进程数据拷贝的所有数据。一旦创建了子进程，由于子进程对这些变量的任何改变都不会反映到父进程的数据中。同样，创建子进程后，对父进程数据的任何改变也不会反映到子进程中。所以，这种类型的进程间通信更像指挥棒传递。一旦父进程传递了某些资源的拷贝，子进程对它的使用就是独立的，必须使用原始传递资源。

在 OS/2 或 NT 等操作系统环境中创建的子进程也从父进程接受一定资源的拷贝。不过, 这些操作系统并不给予子进程父进程文本、数据以及堆栈片断的准确拷贝。子进程所接受资源的拷贝限制于环境数据和文件描述符。在 OS/2 环境中, 子进程同时复制诸如鼠标和视频模式这样的资源。在 OS/2 或 NT 环境中, 所有的对话都有一些由子进程复制的缺省环境变量。在 UNIX 环境中, 所有的 shell 都有一些被子进程继承的基本环境变量。表 5-3 列出了在 UNIX 和 OS/2 环境中共有的一些环境变量 (请注意, 可以在运行时创建用户自定义环境变量, 而且可用于路径搜索、常量与参数传递等)。

表 5-3 UNIX 和 OS/2 环境中共有环境变量

环 境	环 境 变 量	描 述
UNIX	\$PATH	搜索目录列表
	\$USER	用户 ID
	\$HOME	起始目录的完整路径名
	\$TERM	进程的终端类型
OS/2	PATH	.exe、.cmd 的搜索位置
	DPATH	用于设置地区、数据路径
	LIBPATH	DLL 搜索路径
	COMSPEC	用于查找命令解释器的位置

二、命令行参数

通过命令行参数 (command-line argument) 可以完成另一种类型的单向、一次性进程间通信。命令行参数在调用一个 exec 或派生调用操作系统时传递给子进程。命令行参数通常在其中一个参数中作为 NULL 终止字符串传递给 exec 或派生函数调用。大部分 OS/2 和 NT 环境中的 C++ 编译器通过库调用支持 exec 或派生函数家族。UNIX 环境中的 C++ 编译器支持 exec 函数家族。表 5-4 列出了共同的 exec 和派生函数以及它们的意义。这些函数可用于按单向、一次性方式给予进程传递值。

表 5-4 共同的 exec 和派生函数及其意义

函数调用	描 述
execl()	传递指定为完整路径名的程序路径名, 或者当前目录的相对路径, 将命令行参数作为一个列表和 NULL 指针传递。子进程从父进程继承环境
execlp()	与 execl() 参数相同, 但使用环境变量来搜索可执行文件。子进程从父进程继承环境
execle()	与 execl() 参数相同, 但还有一个表示新环境的额外参数
execv()	传递路径名和参数数组
execvp()	传递一个从环境变量中所发现参数和前缀构建的路径名
execve()	传递一个路径名、命令行参数数组以及环境参数数组

例如，程序清单 5-1 中的程序应有 5 个参数。这些参数保存在 `argv[]` 中。第一个参数 `argv[0]` 包含可执行文件的名字，在这里是 `list5-1.exe`。剩下的 4 个参数是整数。这些整数将用于构建 `mt_rational` 对象。一旦构建了对象，就将它们相加并发送到 `stdout`。作为一个子进程来执行 `list5-1.exe`，同时，执行时需要来自父进程的进程间通信。这里需要的进程间通信是一个简单的单向、一次性通信。程序清单 5-2 中的程序使用其中一个 `exec` 函数来给程序清单 5-1 传递必需的命令行参数。

程序清单 5-1 演示命令行参数的使用

```
#include <iostream.h>
#include "mtration.h"
#include <stdlib.h>

void main(int argc, char *argv[])
{
    if(argc == 5){
        try{
            mt_rational X(atol(argv[1]),atol(argv[2]));
            mt_rational Y(atol(argv[3]),atol(argv[4]));
            cout << X + Y;
        }
        catch(general_exception &X)
        {
            cout << X.message() << endl;
        }
    }
}
```

程序清单 5-2 演示使用 `execl`，它是用于派生子进程 `exec()` 函数家族之一

```
#include <iostream.h>
#include <strstream.h>
#include <stdlib.h>
#include <process.h>

void main(int argc, char *argv[])
{
    if(argc == 5){
        char Argument[81] = "";
        strstream Buffer;
        Buffer << (atol(argv[1]) * 2) << " ";
        Buffer << (atol(argv[2]) * 3) << " ";
        Buffer << (atol(argv[3]) + atol(argv[4])) << " ";
    }
}
```



```

    Buffer << atol(argv[4]) << ends;
    Buffer.getline(Argument,80,'\0');
    cout << "Result " << Argument << endl;
    execl("list5-1.exe","List5-1.exe",Argument,NULL);
}

```

程序清单 5-2 中的程序还接受保存在 `argv[]` 中的 5 个命令行参数。第一个参数 `argv[0]` 包含程序的名字, 在这里为 `list5-2.exe`。剩下的参数为 4 个整数。该程序将第一个整数乘以 2, 第二个整数乘以 3。将第一个和第三个整数相加。处理每个整数时, 同时将它们插入和格式化, 放到 `strstream` 对象中。最后, 从 `strstream` 对象中提取命令行参数并保存在 `Argument` 中。此时, `Argument` 包含即将通过 `execl()` 调用传递给程序清单 5-1 中程序的 4 个整数。程序清单 5-2 中的程序为程序清单 5-1 中程序的父进程, 而且使用了一个命令行参数进行简单进程间通信。

三、管道

继承资源以及命令行参数是最简单形式的进程间通信。它们同时有两个主要限制。除了文件描述符外, 继承资源是 IPC 的单向、一次性形式。传递命令行参数也是单向、一次性 IPC 方法。这些方法也只限制于关联进程。如果进程不关联, 命令行参数和继承资源不能使用。还有另一种结构, 称做管道 (pipe), 它可用于在关联进程间以及无关联进程间进行通信。

图 5-5 显示了两个进程 A 和 B 的地址映射。地址映射没有重叠。每个进程都有自己的私有地址空间。进程 A 封装并拥有自己的文本、堆栈以及数据片断。进程 B 同样如此。操作系统的一个主要工作是阻止进程访问它们并不拥有的地址空间或资源。如果进程 A 试图写或访问进程 B 的数据片断, 就发生一个错误。这个错误通常表现为片断错误或页保护错误。图 5-5 中进程间边界由操作系统提供, 而且表示访问策略。但也有时候, 进程希望提供访问数据片断、堆栈片断或堆存储中值的权限。这些值可以使用管道在进程间传递。

1. 什么是管道

管道是一种数据结构, 像一个序列化文件一样访问。它形成了两个进程间的一种通信渠道。管道结构通过使用文件和写方式来访问。如果进程 A 希望通过管道发送数据给进程 B, 那么进程 A 向管道写入数据。为了让进程 B 接收此数据, 进程 B 必须读取管道。与命令行参数形式的 IPC 不一样, 管道可以用于双向通信。在图 5-6 中, 进程 A 和进程 B 之间的数据流是双向通信的。

注意, 有两种管道。进程 A 向管道写入数据, 从另一个管道读取数据。写入的数据以及从管道中读取的数据由一系列字节简单表示。在管道两边, 这些数据的意义取决于程序。管道可用于在进程间发送单个字节或大量字节。命令行参数以及大部分继承资源形成了 IPC 的一种一次性类型, 管道可以在程序的整个执行期间使用, 在进程间发送和接收数据。所以, 管道充当可访问管道的进程间的一种可活链接。有两种基本管道类型:

- 匿名管道 (anonymous pipe);
- 命名管道 (named pipe)。

只有关联进程可以使用匿名管道来通信。无关联进程必须使用命名管道。

2. 匿名管道如何工作

通过文件描述符或文件句柄提供对匿名管道的访问。对系统 API 的调用创建一个管道，并返回一个文件描述符。这个文件描述符于是用作 `read()` 或 `write()` 函数的一个参数。当通过文件描述符调用 `read()` 或 `write()` 函数时，数据的源和目标就是管道。例如，在 OS/2 环境中，使用系统调用 `DosCreatePipe()` 创建匿名管道：

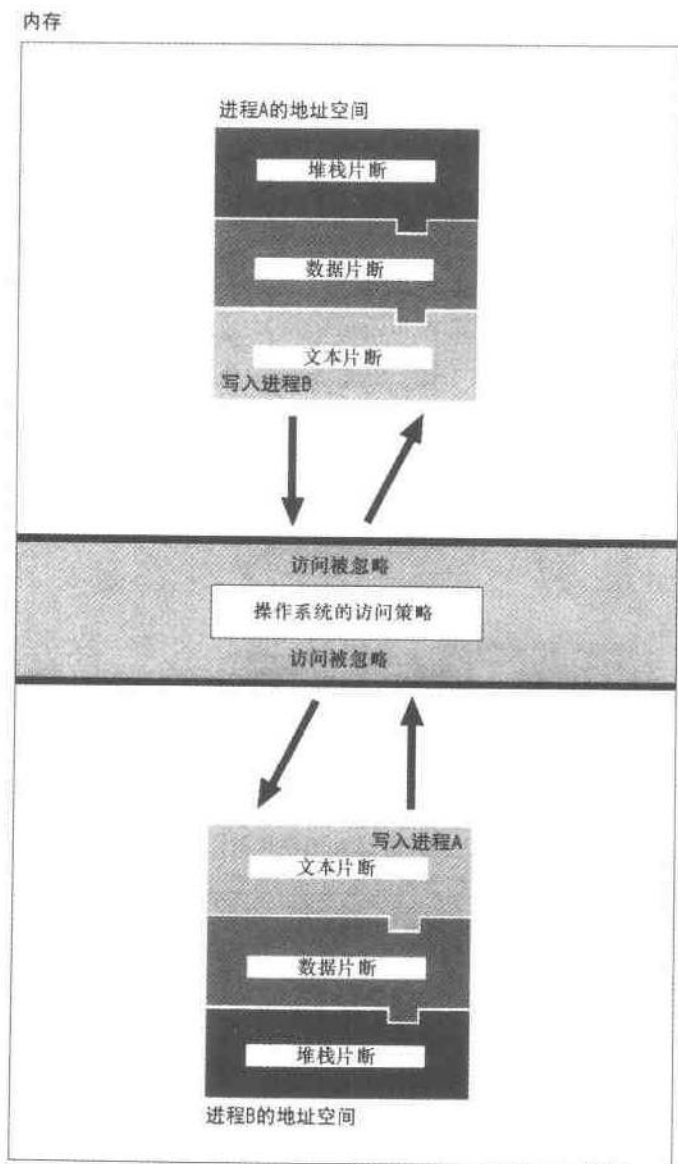


图 5-5 两个进程 A 和 B 的地址映射。每个进程都有自己的私有地址空间。进程间边界由操作系统提供，并表示访问策略。这些访问策略阻止进程访问它们并不拥有的地址空间或资源

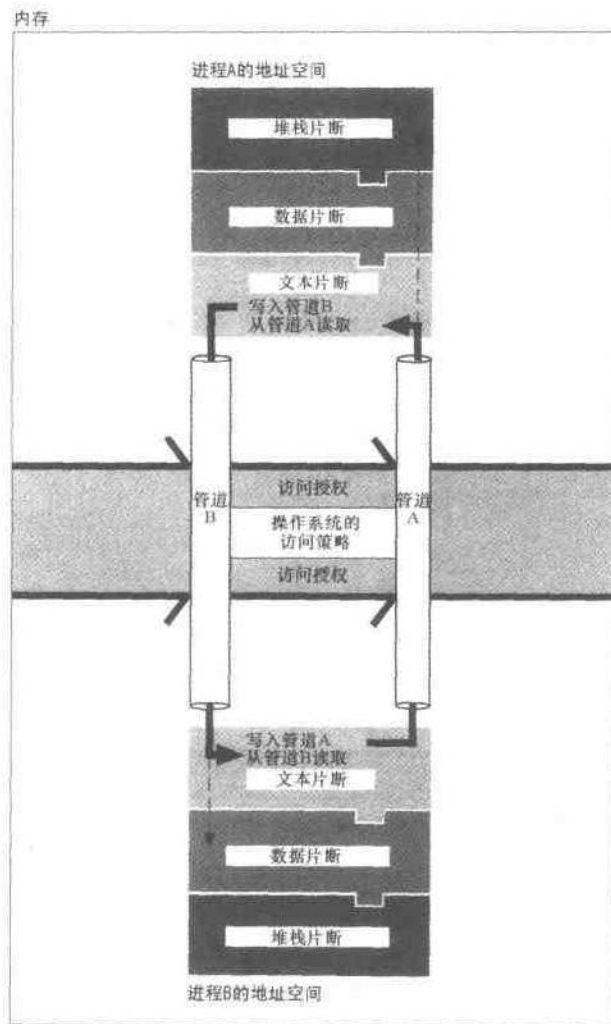


图 5-6 演示进程间的管道结构。进程 A 通过写入管道 B 发送数据给进程 B。
进程 B 从管道 B 中读取数据。进程 B 通过写入管道 A 发送数据给进程 A。
进程 A 从管道 A 中读取数据。进程 A 和进程 B 间的数据是双向通信的

```
void main(void)
{
    PFHILE ReadHandle;
    PFHILE WriteHandle;
    DosCreatePipe(ReadHandle, WriteHandle, Size);
    ...
    ...
    ...
}
```

`ReadHandle` 和 `WriteHandle` 是两个指向文件句柄的指针。`ReadHandle` 用于管道的读访问，`WriteHandle` 用于管道的写访问。匿名管道可以用于关联进程间的通信，因为匿名管道是借助于文件描述符实现的，同时子进程从父进程继承了文件描述符。因为 `DosCreatePipe()` 函数建立了两个文件描述符，任何创建了管道的子进程都可以访问这些文件描述符。当父进程向管道写入数据时，子进程可以读此数据，因为子进程可以访问父进程的文件描述符。当子进程写数据到管道时，父进程可以读此数据，因为父进程拥有子进程所继承的文件描述符。

让我们看一个向匿名管道写入数据的简单例子，并从一个匿名管道中读取数据。与进程间通信的命令行方法不一样，管道允许我们执行双向通信。同时，管道进程创建的管道可以多次使用，而且可以相关进程的整个执行期间存在。在这个例子中，父进程将向管道写入数字，然后子进程读取该数字，处理它们，接着将它们写回管道，让父进程访问它们。程序清单 5-3 以及 5-4 中的程序显示了两个进程间匿名管道的创建和用法。程序清单 5-3 是父进程，程序清单 5-4 是子进程。每个进程执行独立的任务。序清单 5-3 中的程序执行 `pipes.exe`。序清单 5-4 中的程序执行 `unpipe.exe`。然后父进程使用 `DosExecPgm()` 来派生一个子进程。因为子进程继承了父进程的文件描述符，所以子进程可以访问 `InPipe` 和 `OutPipe` 文件描述符。

程序清单 5-3 演示使用匿名管道来重定向子进程的标准输入与标准输出

```
#define INCL_DOSQUEUES
#define INCL_DOSPROCESS
#include "mtration.h"
#include <fstream.h>

const int PipeSz = 256;
char Buffer[PipeSz] = "";
const int Sout = 1;
const int Stin = 0;
HFILE InPipe, OutPipe;

HFILE TempOut;
HFILE TempIn;
HFILE Keep;
HFILE KeepIn;
char ErrorMessage[128] = "";

RESULTCODES Result;

ULONG NumBytesRead;
ULONG NumBytes;

mt_rational Number(1,1);
```

```

void main(void)
{
    DosDupHandle(Sout, &Keep);
    DosDupHandle(Stin, &KeepIn);
    DosCreatePipe(&InPipe, &OutPipe, PipeSz);
    DosDupHandle(OutPipe, &TempOut);
    DosDupHandle(InPipe, &TempIn);
    DosExecPgm(ErrorMessage, sizeof(ErrorMessage), EXEC_ASYNC,
    (PSZ) NULL, (PSZ) NULL, &Result, "unpipe.exe");
    DosWrite(OutPipe, "5", sizeof(int), &NumBytes);
    DosWrite(OutPipe, "4", sizeof(int), &NumBytes);
    DosDupHandle(Keep, &TempOut);
    DosRead(InPipe, Buffer, sizeof(int), &NumBytesRead);
    cout << Buffer << endl;
    DosRead(InPipe, Buffer, sizeof(int), &NumBytesRead);
    cout << Buffer << endl;
    DosClose(InPipe);
    DosClose(OutPipe);
}

```

如果变量名字 `InPipe` 和 `OutPipe` 没有按某种方式传递给子进程，子进程将怎么办呢？父进程控制着子进程的标准输入和标准输出文件描述符。父进程使用这种控制权重新分配子进程的标准输入和标准输出，让它们与文件句柄 `InPipe` 和 `OutPipe` 连接。在程序清单 5-3 中，使用 `DosDupHandle()` 函数来完成这一点。子进程的缺省 I/O 渠道已经连接到了管道。所以，当子进程读取它的标准输入或标准输出时，它实际上是访问父进程创建的管道。程序清单 5-3 中的父进程只是发送数字 5 和 4 到管道。子进程读取管道并通过 `cout` 对象发送数字到标准输出。虽然子进程发送数字到标准输出，它们实际上到达管道。父进程从管道读取数字并打印出来。程序清单 5-4 中的子进程向 `cout` 对象中插入从父进程接收的数字，`cout` 对象与标准输出连接。

程序清单 5-4 程序清单 5-3 中程序的子进程。标准输入和标准输出被父进程重定向

```

#define INCL_DOSQUEUES
#define INCL_DOSPROCESS
#include <os2.h>
#include <fstream.h>

ULONG NumBytesRead;
char Buffer1[2] = "";
char Buffer2[2] = "";
const int StIn = 0;

```

```

void main(void)
{

```

```

DosRead(StIn, Buffer1, 1, &NumBytesRead);
DosRead(StIn, Buffer2, 1, &NumBytesRead);
cout << Buffer1 << Buffer2;
}

```

匿名管道通常用于重定向子进程的标准输入和标准输出。管道形成了一个循环缓冲器，它允许父进程读取子进程的标准输出。因为这些管道是匿名的，所以它们只可以用作关联进程间的通信渠道。如果进程是关联的，匿名管道可用于连接标准输入和标准输出，形成“生产者”和“消费者”关系链，在这种关系中，只是从一个 `stdin` 到下一个 `stdout` 沿着管道发送数据。

3. 无关联进程间的管道（命名管道）

将管道用作两个无关联进程间的通信渠道，程序员必须使用命名管道，它可以看作一种具有某名字的特殊类型文件。进程可以根据它的名字访问这个管道。通过匿名管道，父和子进程可以单独使用文件描述符来访问它们所共享的管道，因为子进程继承了父进程的文件描述符，同时文件描述符用作 `read()` 或 `write()` 函数的参数。因为无关联进程不能访问彼此的文件描述符，所以不能使用匿名管道。由于命名管道提供该管道的一个等价文件名，任何知道此管道名字的进程都可以访问它。下面是命名管道相对于匿名管道的优点：

- 命名管道可以被无关联进程使用。
- 命名管道可以持久。创建它的程序退出后，它们仍然可以存在。
- 命名管道可以在网络或分布环境中使用。
- 命名管道容易用于多对一关系中。

与访问匿名管道一样，命名管道也是通过 `read()` 或 `write()` 函数来访问。两者之间的主要区别在于命名管道的创建方式以及谁可以访问它们。命名管道可以用于建立一个进程间通信的客户/服务器模型。访问命名管道的进程可能都位于同一台机器上，或位于通过网络通信的不同机器上。由于管道的名字可以包含通过管道所在服务器的逻辑名，所以能够跨网络访问管道。例如，`//Server1//PIPE//MyPipe` 可以作为一个管道名字。假如 `Server1` 为网络服务器的名字。当打开或访问这个管道的调用解析该文件名时，首先定位 `Server1`，然后访问 `MyPipe`。NT 和 OS/2 都支持允许跨网络访问的命名管道。创建命名管道需要使用一个系统 API 调用。表 5-5 在 OS/2 和 NT 环境中命名管道工作所需的 API 调用。

表 5-5 OS/2 和 NT 环境中命名管道工作所需的 API 调用

环 境	函 数	描 述
OS/2	<code>DosCallNPipe()</code>	写入双重消息管道、从中读取，然后关闭该管道
	<code>DosCreateNPipe()</code>	创建一个命名管道
	<code>DosDisconnectNPipe()</code>	客户进程已经关闭命名管道的通知
	<code>DosConnectNPipe()</code>	通过将管道设置成侦听状态，为客户进程准备一个命名管道
	<code>DosPeekNPipe()</code>	检查命名管道中的数据，而不从管道中取出数据。它返回管道状态的信息

续表

环 境	函 数	描 述
OS/2	DosQueryNPipeInfo()	返回命名管道的信息
	DosQueryNPipeSemState()	返回附加于一个 muxwait 信号量和一个共享事件上的本地命名管道的信息
	DosSetNPHState()	重置命名管道的实际阻塞模式
	DosSetNPipeSem()	在本地命名管道上附加一个共享事件信号量
	DosTransactNPipe()	写入双重消息管道, 然后从管道中读取
	DosWaitPipe()	等待命名管道的某个实例变成可用
Win32	CallNamedPipe()	连接到一个管道实例, 然后写入一条消息、读取消息, 并关闭管道句柄。它只能被客户进程和消息管道使用
	PeekNamedPipe()	读取通过管道传输的数据, 而不会从字节管道或消息管道取出数据。它返回管道实例的信息
	DisconnectNamedPipe()	客户和进程使用完管道实例后, 服务器进程调用此函数。它关闭到客户进程的连接。客户句柄变成无效, 并抛弃管道中的所有未读数据
	TransactNamedPipe()	写请求消息, 并读答复消息。如果将调用进程的管道句柄设置成消息读取模式, 就可以与消息管道一起使用
	FlushFileBuffers()	服务器进程调用该函数确保客户进程读取写入管道的所有字节和消息。如果函数没有返回到客户进程, 它就从管道读取了所有的数据
	GetNamedPipeInfo()	获取命名管道的信息。它返回管道的类型、输入和输出缓冲器的大小, 以及可以创建的管道实例的最大数
	GetNamedPipeHandleState()	提供句柄, 包括管道的读和等待模式、管道实例的当前数, 以及通过网络进行通信的管道的其它所有重要信息
	SetNamedPipeHandleState()	设置管道句柄的读和等待模式。它还控制搜集字节的最大数, 或者客户进程与远程服务器通信时传输消息前等待的最长时间
	ReadFile() WriteFile()	用于字节和消息管道。使用一个事件对象来通知其完成

一旦创建了命名管道, 它们可以被任何知道该管道名的进程打开。我们可以修改程序清单 5-3 和 5-4 中的程序代码, 使用命名管道而不是使用匿名管道。

程序清单 5-5 包含服务器程序 `npipes.exe`。当使用命名管道时，服务器为创建命名管道的程序。所有访问管道的其它进程都称做客户。程序清单 5-6 包含一个客户程序 `clnpipe.exe`。

程序清单 5-5 演示进程间通信对命名管道的使用。进程充当服务器

```
#define INCL_DOSNMPIPES
#define INCL_DOSPROCESS
#include <os2.h>
#include <fstream.h>
#include <string.h>

const int PipeSz = 256;
char PipeName[40] = "";
char Buffer[PipeSz] = "";
HFILE PipeHandle;
unsigned long PipeMode;
unsigned long OMode;
unsigned long Duration;
unsigned long NumBytes;
unsigned long NumBytesRead;
char Problem[PipeSz];
RESULTCODES Result;

void main(void)
{
    OMode = NP_ACCESS_DUPLEX;
    PipeMode = NP_WMESG | NP_RMESG | 0x01;
    Duration = 20000;
    strcpy(PipeName, "\\PIPE\\PIPE1");
    DosCreateNPipe(PipeName, &PipeHandle, OMode, PipeMode,
255, 255, Duration);
    DosExecPgm(Problem, PipeSz, EXEC_ASYNC, NULL, NULL, &Result,
"CLNPIPE.exe");
    DosConnectNPipe(PipeHandle);
    DosWrite(PipeHandle, "5", 1, &NumBytes);
    DosWrite(PipeHandle, "4", 1, &NumBytes);
    DosRead(PipeHandle, Buffer, 1, &NumBytesRead);
    cout << "Server Read: " << Buffer << endl;
    DosRead(PipeHandle, Buffer, 1, &NumBytesRead);
    cout << "Server Read: " << Buffer << endl;
    DosClose(PipeHandle);
}
```


程序清单 5-6 使用命名管道的客户进程执行进程间通信

```
#define INCL_DOSQUEUES
#include <os2.h>
#include <fstream.h>
#include <string.h>

char PipeName[40] = "";
HFILE PipeHandle, PipeOut;
unsigned long NumBytesRead;
unsigned long Action;
char Buffer1[2] = "";
char Buffer2[2] = "";

void main(void)
{
    strcpy(PipeName, "\\PIPE\\PIPE1");
    DosOpen(PipeName, &PipeHandle, &Action, 0, FILE_NORMAL,
        FILE_OPEN, OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
        (PEAOP) NULL);
    DosRead(PipeHandle, Buffer1, 1, &NumBytesRead);
    DosRead(PipeHandle, Buffer2, 1, &NumBytesRead);
    cout << "Client Read " << Buffer1 << " " << Buffer2 << endl;
    DosWrite(PipeHandle, Buffer1, 1, &NumBytesRead);
    DosWrite(PipeHandle, Buffer2, 1, &NumBytesRead);
}
```

服务器程序负责建立客户与服务器进程间管道通信的协议。服务器程序必须指定访问模式、阻塞模式、管道类型、读取模式、缓冲器大小以及管道使用的即时计数。在程序清单 5-5 的程序中，服务器建立管道来发送和接收数据。服务器指定这个管道将处于阻塞模式。这意味着，当请求管道的 `read()` 或 `write()` 时，在一定的条件下发生阻塞。如果请求 `read()`，而且管道是空的，则 `read()` 阻塞，直到数据可以被读取为止。如果请求的是管道的 `write()`，而且管道已满，则 `write()` 阻塞，直到管道中腾出了更多空间放入数据为止。`DosCreateNPipe()` 调用中的 `Omode` 和 `Pipemode` 参数决定数据流以及阻塞或非阻塞读与写。表 5-6 列出了创建命名管道时，服务器的负责工作。

表 5-6 创建命名管道时服务器的责任

服务器责任	描 述
管道创建	服务器进程创建管道
管道连接	服务器进程将自己连接到管道的一个终端
设置继承模式	服务器进程指定子进程是否继承命名管道

续表

服务器责任	描 述
设置访问模式	服务器进程指定访问模式。当服务器创建管道时，它指定数据流经管道的方向。如果它将从客户进程读数据，则指定一个入站管道。如果是写入客户进程数据，则指定一个出站管道。如果是既读又写客户进程，则指定一个双重管道
设置阻塞模式	服务器进程指定一种等待模式，在管道中的数据可用之前，以此模式阻塞读取，而且管道中没有足够的空间保存整个数据缓冲器前，也一直阻塞写。它可以指定一种非等待模式，按这种模式，从一个空管道中读，或者写入一个充满管道都会立即返回一个错误
设置管道类型	服务器进程决定数据流写入管道的形式。管道可以是字节管道，其中服务器和客户将数据写为字节流。管道可以是消息管道，其中服务器和客户将数据写作消息流。服务器和客户定义消息的大小和格式
设置读模式	服务器进程决定如何从管道中读取信息。如果管道是一个消息管道，则数据可以读作消息或字节。如果管道为一个字节管道，则数据只能读作字节
设置管道模式	服务器进程决定创建时服务器终端管道的读模式
设置缓冲器计数	当指定了缓冲器的大小时，服务器进程可以控制缓冲器一次能容纳的消息数量。管道必须处于消息读（message-read）模式，而且必须已知消息的大小
设置实例计数	服务器进程指定可以创建的使用管道实例最大数。管道的实例是具有唯一句柄的带有缓冲器的单独管道
管道断开连接	服务器进程可以客户进程与管道的连接。客户进程在服务器进程断开管道连接前关闭管道。如果客户没有关闭管道，服务器关闭管道，如果客户试图访问此管道，则客户收到错误。客户有机会读取数据前，强迫管道关闭可能导致数据丢失

命名管道不仅可用于无关联进程间、位于不同机器上的两进程间的通信，而且可用于多对一通信。可以建立服务器进程，允许同时通过多个客户访问命名管道。命名管道常常用于多线程服务器。

四、共享内存

共享内存也可以用于实现进程间通信。进程需要可以被其它进程浏览的内存块。希望访问这个内存块的其它进程必须请求对它的访问，或由创建它的进程授予访问内存块的权限。可以访问特定共享内存块的所有进程对它具有即时可见性。共享内存被映射到使用它的每个进程的地址空间。所以，它看起来像另一个在进程内声明的变量。当一个进程写共享内存，所有的进程都立即知道写入的内容，而且可以访问。

进程间共享内存的关系与函数间全局变量的关系相似。程序中的所有函数都可以使用全局变量的值。同样，共享内存块可以被正在执行的所有进程访问。内存块可能存在于虚拟存储器中，内存块也可能被映射到特定的物理内存页。也就是说，进程可能共享一个逻辑地址，进程也可以共享某些物理地址。

共享内存块的创建必须由一个系统 API 调用来完成。在 Win32 环境中, 使用 `CreateFile Mapping()`、`MapViewOfFile()` 以及 `MapViewOfFileEx()` API 能很好地完成。图 5-7 显示了 Win32 环境的虚拟内存映射。

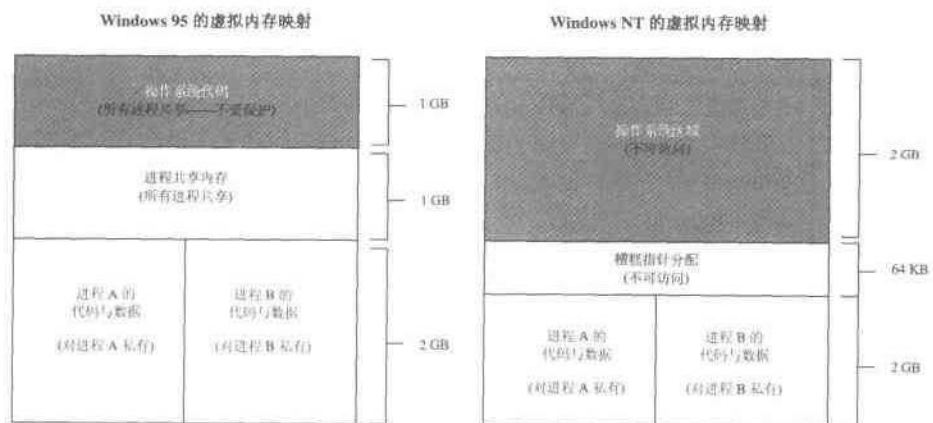


图 5-7 Win32 环境的虚拟内存映射

共享内存块分配位于 Win32 系统中 2~3 GB 地址范围内。一旦调用 `MapViewOfFile()` 和 `MapViewOfFileEx()`, 共享文件映射对象的所有进程都可以立即访问此内存块, 而且在需要时, 可以读和写此内存块。

在 OS/2 环境中, 共享内存通过 `DosAllocSharedMem()` API 来分配。可以访问共享内存的每个进程在它的虚拟内存地址的同一位置可以看到这个块。图 5-8 描述了在 OS/2 环境虚拟内存映射中的共享内存。

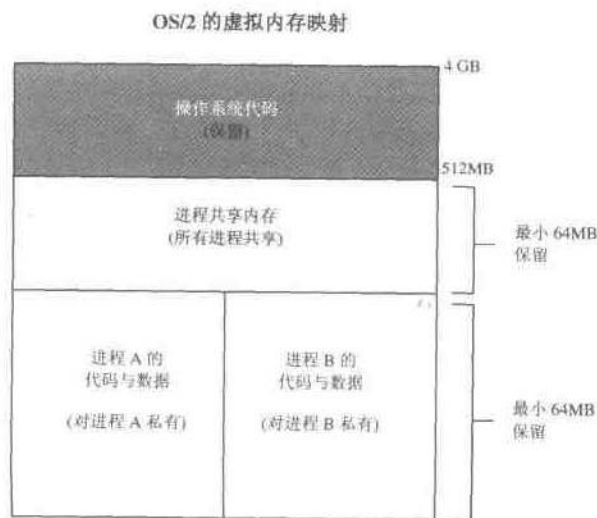


图 5-8 OS/2 环境的虚拟内存映射

在 OS/2 环境中, 共享内存可以是匿名的, 也可以是有名的。如果共享内存为匿名的, 那么必须使用某些形式的进程间通信来给希望应用共享内存的其它进程传递共享内存地址。如果内存是有名的, 则没有必要在进程间形成通信渠道。其它进程可以通过名字访问共享内存。

尽管 OS/2 和 Win32 环境映射 4 GB 空间的情况不一样, 但两者都赋予了程序员一个支持共享内存块的专门区域。通常使用共享内存比使用管道或队列更简单, 也更有效。共享内存块可以用于保存大数据结构, 这些数据结构然后可以从任何数量的进程中有效访问。这种内存可以充当前面向对象数据库、集合对象或容器对象的一种永久存储空间。

共享内存块也可以用于映射文件到内存。按这种方式使用共享内存块减轻了应用程序需要常规文件访问的 I/O 操作代价。表 5-7 列出了在 Win32 和 OS/2 环境中创建和使用共享内存所需的 API。

表 5-7 在 Win32 和 OS/2 环境中创建和使用共享内存所需的 API

环 境	共享内存 API	描 述
OS/2	DosAllocSharedMem()	在虚拟进程地址空间中分配共享内存对象
	DosGetNamedSharedMem()	获取对已存在命名共享内存对象的访问权限
	DosGetSharedMem()	获取对已存在共享内存对象的访问权限
	DosGiveSaredMem()	赋予目标进程访问已存在共享内存对象的权限
Win32	CreateFileMapping()	创建一个文件映射对象; 对文件无限制
	OpenFileMapping()	获取映射对象的句柄
	MapViewOfFile()	获取共享内存的起始地址

五、动态数据交换

动态数据交换 (dynamic data exchange) 是当今可用的进程间通信最强大和完善的形式之一。动态数据交换使用消息传递、共享内存、事务协议、客户/服务器范例、同步规则以及会话协议来让数据和控制信息在进程间流动。动态数据交换对话 (dynamic data exchange session, DDE) 的基本模型是客户/服务器。服务器对来自客户的数据或动作做出反应。客户和服务器可以以多种关系来通信。最常见的关系如图 5-9 所示。

一个服务器可以与任意数量的客户通信。一个客户也可以与任意数量的服务器通信。单个 DDE 代理既可以是客户, 也可以是服务器。也就是说, 进程可以从一个正为另一个进程执行服务的 DDE 代理请求服务。

进程间通信的 DDE 形式使用消息系统来完成, 消息系统是由操作系统提供的。在所有 DDE 交互中主要有 4 种组件。图 5-10 是对这 4 种主要组件的高层概览。

这 4 种组件的结合提供了进程间通信能力, 这种能力是到目前为止所讨论的技术不能相比的。客户和服务器均代表系统内的进程。客户间的会话组成了通信的实质以及进程间传递的内容。进程间的共享内存用于保存任何种类的数据, 从简单数据类型到复杂、面向对象数据库。4 个主要组件可以用于局部进程间通信以及跨网络进程间的通信。

1. DDE 服务器

DDE 服务器至少有 3 个基本组件：

- 应用；
- 主题；
- 项。

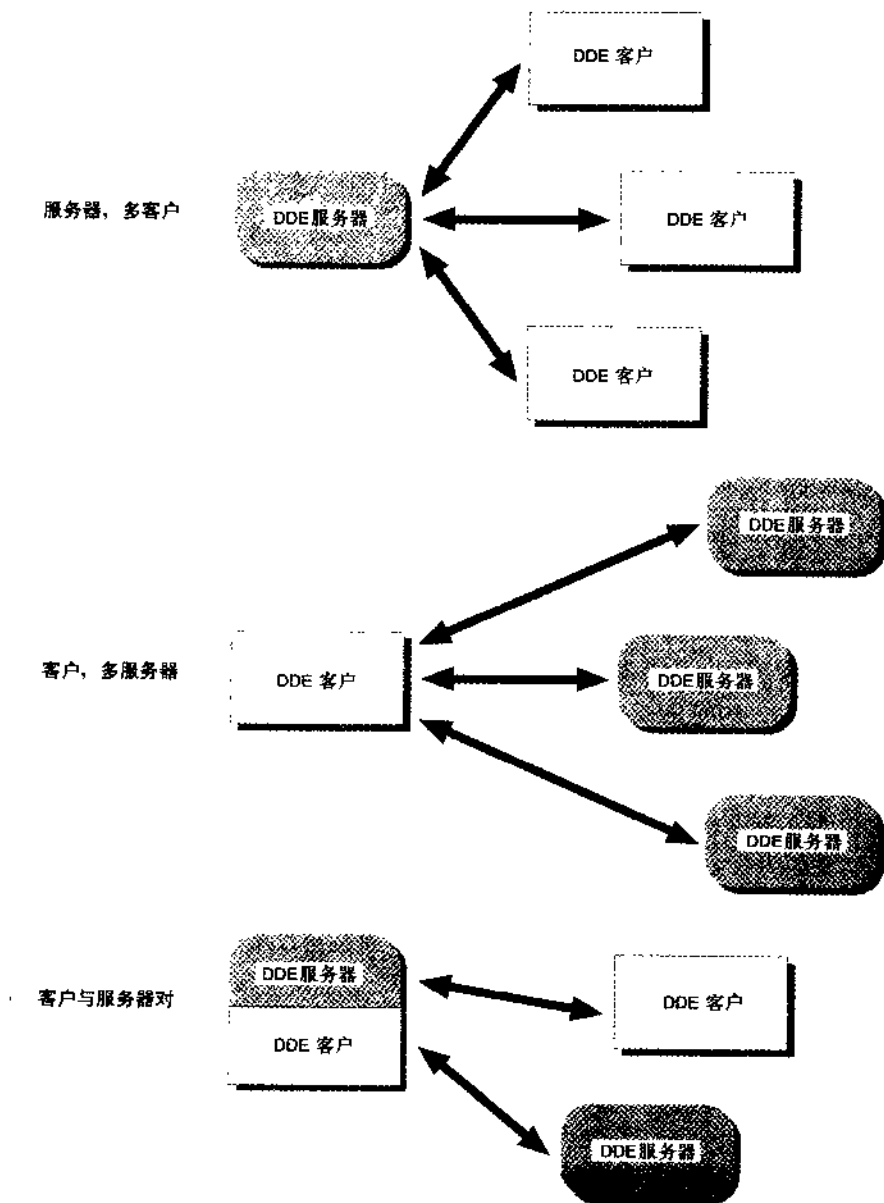


图 5-9 DDE 服务器和客户的可能模型

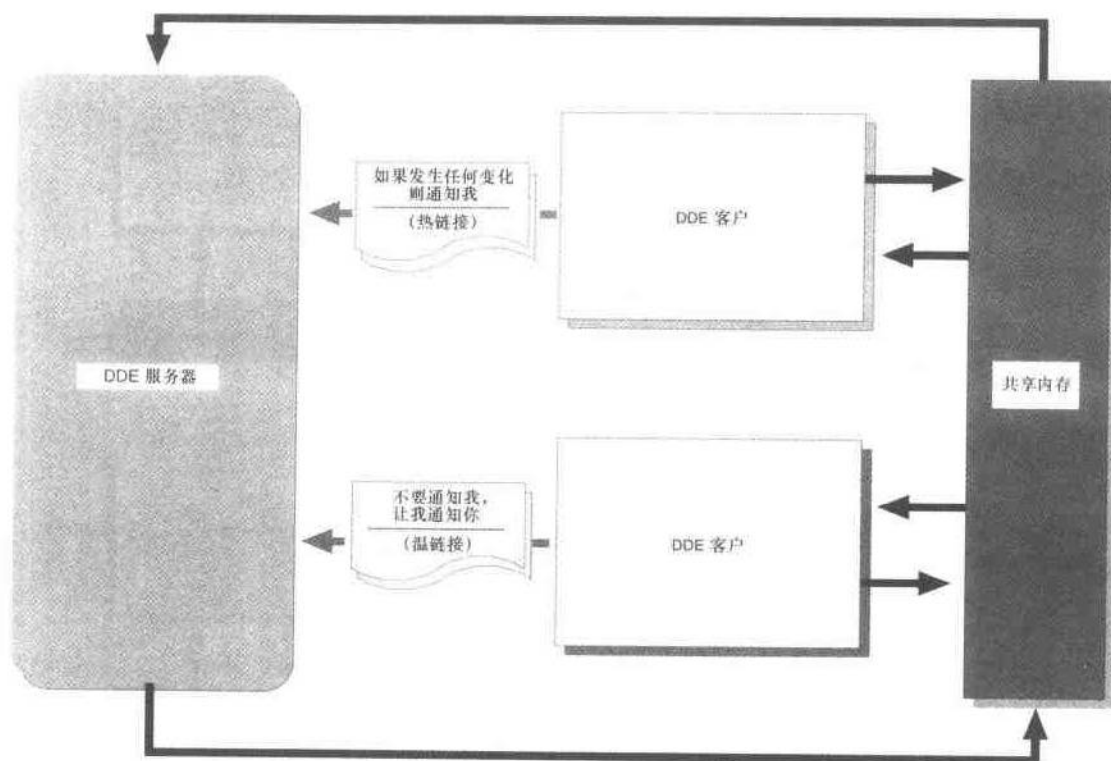


图 5-10 DDE 交互的主要组件的高层概览。主要组件为 DDE 服务器、DDE 客户、共享内存以及链接

应用指服务器的名字。每个 DDE 服务器都有一个单个的名字，而且至少有一个主题，它指 DDE 服务器知道的一般性主旨。一个服务器可能有多个主题，而且每个主题至少有一个项。一个主题可能包含多个项。项指与主题相关的特定数据片断。例如一个多媒体服务器，名叫 media server，它可能有多个主题。一个主题为设备容量，另一个主题为多媒体文件格式，它具有像 wave、midi、mpeg 或 avi 等之类的项。media server 可以分析它所在的多媒体环境，以及它所使用的任何多媒体元素。media server 的工作是标识和评价多媒体设备、方案以及数据元素。应用、主题和项的标识如图 5-11 所示。

应用就是 media server。media server 具有的主题为多媒体设备容量以及多媒体文件格式。请注意，每个主题都有它支持的一系列项。服务器可以对请求 media server 的客户作出反应，或者询问与多媒体设备容量以及多媒体文件格式相关的主题。

一般而言，服务器从查询主题的客户那里接收到消息。服务器发出一条肯定答复，表明它知道这个主题，否则发出一条否定消息。同样，一旦对主题达成一致，客户将发送一条与该主题内特定项有关的消息。如果服务器有这些项，它必定发送一条肯定答复。如果服务器没有这些项，则发送否定答复。DDE 客户与服务器间的交互模式是基本的。服务器必须既有主题，也有该主题内的某个项，才对客户有用。根据情况，服务器必须发送肯定或否定答复。

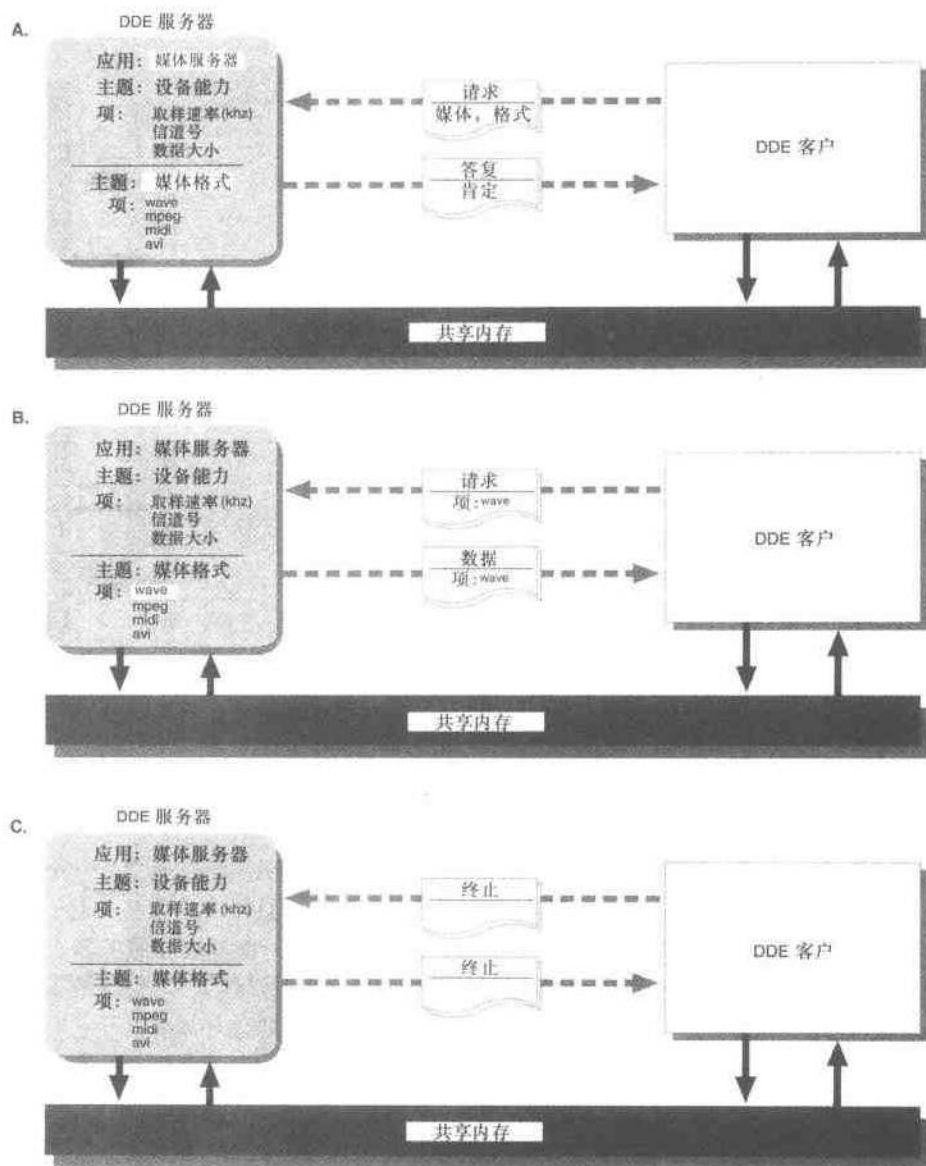


图 5-11 应用、主题和项的标识

2. DDE 客户

在 DDE 事务中的会话由客户启动。客户可以发送一条应用消息。如果应用消息包含一个名字，那么有此名字的任何服务器都必须给客户发送一条肯定答复。客户可以发送一条空应用消息。当发送一条空应用消息时，所有的 DDE 服务器，不论其应用名字是什么，都必须给客户发送一条肯定答复。通过使用空应用消息，客户可以判断所有可用的 DDE 服务器。客户也可以发送

条主题消息。任何知道指定主题的 DDE 服务器都必须给客户发送一条肯定答复。通过应用消息，客户也可以发送一条空主题消息。当发送一条空主题消息时，任何有此主题的 DDE 服务器都必须将消息传达给客户。使用这一技术可以发现所有可用的主题。

根据从 DDE 服务器接收的消息内容，客户必须给服务器发送一条肯定或否定答复。在图 5-11 中服务器与客户之间的基本对话可能包含主题多媒体文件格式的请求。在这种情况下，media server 将作出肯定答复的反应。客户然后将对应的多媒体文件传递给 media server。media server 可能作出肯定答复的反应，并告知数据项 wave 的存在。media server 为客户分析了文件格式。现在客户可以准备适当的设备并播放文件。当客户完成了与服务器的会话时，它将发送一条终止通知，让服务器知道不再有其它请求。

3. DDE 会话

DDE 客户和服务器彼此间发送一系列消息。这一系列消息称做会话 (conversation)。客户启动与一个或多个服务器的会话，然后开始发送和接收消息。发生的会话有两种基本类型：

- 如果发生变化则通知我 (热链接)。
- 不要通知我，让我通知你 (冷链接)。

当客户请求服务器报告考虑之中数据项的所有变化时，就发生第一种类型的会话。例如，如果数据项是一个附加到某多媒体天气地图上的卫星 mpeg 或 jpeg 文件，客户可能希望卫星图像变化时更新地图。假想有一个 Web 浏览器，它与一个多媒体天气预报有一个 DDE 链接，其中有当地飓风、大风雪、洪水、森林火灾等的变化，天气预报随时更新。客户可能请求与服务器的热链接。热链接请求服务器在被请求的数据项发生变化时提醒客户。

第二种类型会话与热链接相比，它是一种更临时性联络的结果。在天气预报例子中，如果我们只对特殊位置的某特殊天或若干天的天气感兴趣，我们就要向服务器发出一次性请求。不需要每分钟更新天气数据。我们可能只希望知道今天的天气情况如何。客户将请求一个冷链接。之所以称做冷链接，是因为一旦客户得到了数据，客户就不会再得到任何后续变化的通知，除非客户发出显示请求。

在这两个基本类型的会话中，可以发生 5 种主要类型的事务 (transaction)：

- 请求 (request)；
- 建议 (advise)；
- 取消建议 (unadvise)；
- 执行 (execute)；
- 发送 (poke)。

“请求”用于要求服务器的数据，通常是基于冷链接。“建议”事务启动一个热链接，请求服务器连续不断地通知客户。“取消建议”事务用信号通知服务器建议条件结束，不再需要热链接。执行事务将导致服务器执行某些任务。使用执行事务，客户可以遥控服务器，从服务器请求数据和动作。

如果服务器不能提供所请求的数据或动作，就给客户发送一个否定的答复。DDE 包含一系列的请求、答复、数据以及命令形式的消息。例如，客户可能请求 media server 显示一个位图文件，或解压一个电影文件。第 5 种事务类型就是发送 (poke)。客户使用“发送”通过服务器来直接存

放数据。

4. 共享内存和 DDE 会话

DDE 客户和服务会话结合使用消息和共享内存来完成。典型情况下, 消息包含指向服务器或客户窗口的指针, 一个数字值表示消息, 一个值表示发送者或接收者是否必须答复消息, 还有一个接收者需要的指向共享内存的指针。根据应当的链接类型(冷或热链接), 完成事务时, 共享内存可以被删除, 也可以不被删除。共享内存可能是小到只有单个字节的内存块, 也可能表示大到数 GB 的位图或电影文件。共享数据的传达手段是给予 DDE 额外的灵活性和能力。

DDE 客户和服务可以针对某范围的主题以及此主题内的项进行会话。一旦对主题-项对(topic-item pair) 达到一致, 就可以在客户进程和服务进程间交换数据。数据几乎可以为所需要的任何形式和大小。相对于其它方法的进程间通信, 应用标识、主题、项以及共享内存的结合实际上使动态数据交换更灵活。在 DDE 服务器的概念上再添加线程更是增加了它作为一个 IPC 组件的有用性。

5.3 线程间通信

也许进程和线程间巨大的区别就是单个进程有自己的地址空间(address space), 而线程没有。线程共享进程的地址空间。当两个进程需要通信时, 它们一般使用某种两进程外部的、两进程都能访问的数据结构来实现。使用这种数据结构传递数据或命令, 两个进程可以通信。当两个线程通信时, 它们一般使用属于同一进程的部分数据结构来实现。进程内的两个线程可以访问同一个数据片断。进程内的两个线程可以相互从堆栈片断传递值。对于需要通信的两个线程, 它们没有使用两个进程通信所使用的进程间通信机制。例如, 线程不会使用管道相互通信。图 5-12 对比了线程使用的通信类型与进程使用的通信类型。

线程间通信类型

如果一个进程内的某线程需要与另一个进程内的某线程进行通信, 所谈论的就不再是线程间通信(interthread communication)。此时的通信涉及多个进程, 程序员需要使用进程间通信机制(interprocess communication mechanism)。

如果只涉及一个进程, 而且这个进程包含相互间需要通信的多个线程, 程序员就需要使用线程间通信了。就像有多种机制支持进程间通信一样, 也有多种机制支持线程间通信。在大部分情况下, 进程间通信的代价高于线程间通信的代价。在进程间通信期间必须由操作系统创建外部数据结构, 这与在进程内创建数据结构相比, 前者需要更多的系统处理时间。操作系统进行的进程创建过程比线程创建过程需要更多的步骤。在需要并发的多种编程场合, 线程创建以及线程间通信机制的效率使得线程成为更具吸引力的方案。表 5-8 列出了用于完成线程间通信的基本机制。

一、全局数据

线程相对于进程的一个重要优点是, 线程可以共享全局变量(global variable)。在 C++ 中函数的外部声明的变量是全局变量。进程内的所有线程可以平等地访问全局变量。如果任何线程更

改了这个变量，则程序内的所有变量都可以立即应用这一变化。程序清单 5-7 中的程序显示了线程间通信的这种用途。

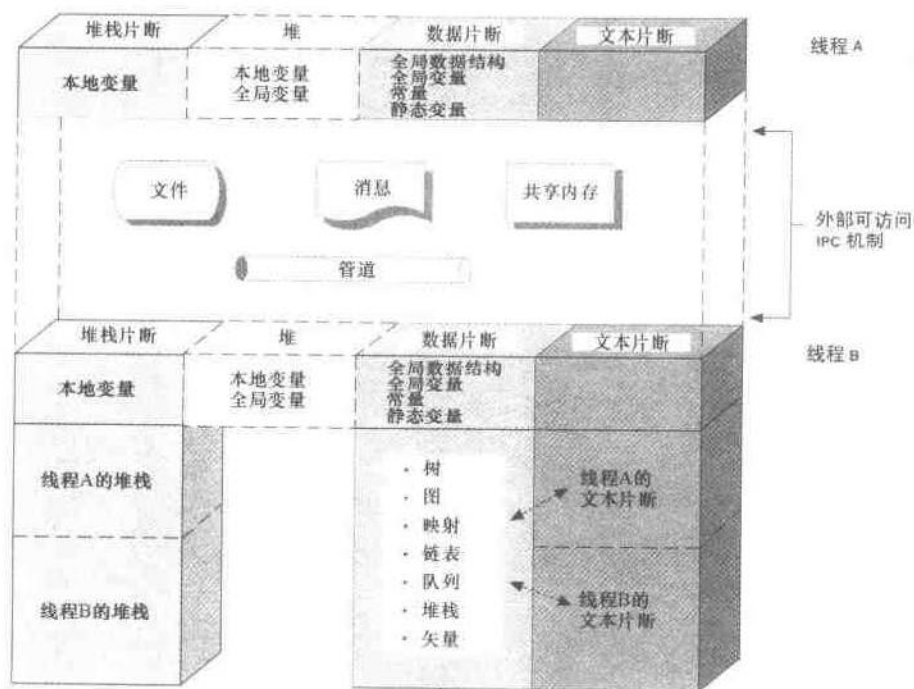


图 5-12 对比线程使用的通信类型与进程使用的通信类型

表 5-8

用于完成线程间通信的基本机制

线程间通信	描 述
全局数据	已经在任何函数外部声明的变量可以被进程中的所有线程平等访问。对进程中所有线程的任何修改都将立即反映到进程中的所有线程中
全局变量	全局声明的变量可以被进程中的任何线程访问。进程中任何线程对该变量的所有修改都将立即反映到该进程中所有线程中
全局数据结构	在所有函数外部声明的数据结构可以被进程中的任何线程访问。这些结构可以是矢量、集、图表、映射、多重映射、多重集、双端队列、树、链表、队列等等
参数	在同一进程中线程间传递的参数。它是一个可以指向任何数据类型的有效指针
文件句柄	线程间共享文件的文件句柄。如果一个线程推进读/写指针，其它访问该文件的所有线程都获得相同的偏移

程序清单 5-7 演示全局变量 Count 的使用，它由两个线程更新

```
#include <iostream.h>
#include <pthread.h>

int Count = 0;

void *threadA(void *X)
{
    Count += 2;
    cout << Count << endl;
}

void *threadB(void *X)
{
    Count += 4;
    cout << Count << endl;
}

void main(void)
{
    pthread_t ThreadA;
    pthread_t ThreadB;
    pthread_create(&ThreadA, NULL, threadA, NULL);
    pthread_create(&ThreadB, NULL, threadB, NULL);
    pthread_join(ThreadA, NULL);
    pthread_join(ThreadB, NULL);
}
```

二、全局变量

程序清单 5-7 中的程序声明了一个全局变量 Count，并将它设置为 0。这个程序还创建了两个线程：threadA()和 threadB()。threadA()将 Count 增加 2，threadB()将 Count 增加 4。因为 Count 已经声明为全局，两个线程都可以访问。threadA()对它所作的任何更改，threadB()立即可以使用，反之亦然。不可能在单个数据片断中由两个单独的进程共享某个变量。如多个进程需要访问同一个变量时，必须由操作系统创建一个共享内存变量。这个共享内存变量是多个进程的外部数据结构。进程 A 数据片断中的任何变量不能被进程 B 修改。同样，进程 B 数据片断中的任何变量也不能被进程 A 修改。在单个进程内的多个线程越过了地址空间的障碍。如果变量声明为全局，进程内的所有线程都可以访问该变量，而且可以不受任何形式的操作系统干预更改变量的值。不需要创建特殊的共享内存数据结构。有时，通过线程访问全局变量的简单性足以证明了使用多线程相对于多进程的优势。

三、全局数据结构

在任何函数外部声明的数据结构也可以被进程内的任何线程访问。尽管进程间通信只支持有限的数据结构集来进行进程间通信，但任何类型的全局数据结构都可以被进程内的线程访问。线程可以使用矢量（vector）、集（set）、图表（graph）、树（tree）、队列（queue）、链表（linked list）、映射（map）、双端队列（deque）、多重映射（multimap）、多重集（multiset），或其它任何集合或容器类来完成线程间通信。当然，如果多个线程用于写入全局数据结构，必须使用合作技术。

程序清单 5-8 中的程序创建了两个线程 threadA() 和 threadB()。程序还创建了 4 个全局变量和 3 个全局数据结构。全局变量包括有理对象 M 和 3 个集迭代器 A、B 和 C。全局数据结构由 3 个 STL 集容器组成：SetA、SetB 和 SetC。程序使用 threadA() 将有理数插入 SetA 和 SetB；threadA() 还使用 STL set_intersection() 算法求 SetA 和 SetB 之间的交集。交集的结果保存在 SetC 中。当 threadA() 进行集运算时，用 threadB 在 SetA 和 SetB 中插入有理数。STL set_union() 算法用于求 SetC 和 SetB 之间的并集（union）。并将集的结果保存在 SetA 中。主线程使用 pthread_join() 等待 threadA() 和 threadB() 的完成。threadA() 和 threadB() 完成后，主线程遍历 SetA、SetB 和 SetC，将它们的成员发送到标准输出。

程序清单 5-8 使用全局 STL 容器作为线程间通信机制

```
#include <iostream.h>
#include <pthread.h>
#include <rational.h>
#include <set.h>
#include <algo.h>

rational M(3,4);
typedef set<rational,rational> rational_set;
typedef set<rational,rational>::iterator rational_iterator;
rational_set SetA,SetB,SetC;
rational_iterator A,B,C;

void *threadA(void *X)
{
    rational Z(5,4);
    rational W(1,8);
    SetA.insert(W);
    SetA.insert(Z);
    set_intersection(SetA.begin(),SetA.end(),SetB.begin(),
                     SetB.end(),inserter(SetC,SetC.begin()),Z);
    SetB.insert(W);
    SetA.erase(SetA.begin(),SetA.end());
}

void *threadB(void *X)
```

```
{  
  
    rational Q(2,3);  
    rational R(5,5);  
    SetB.insert(Q);  
    SetB.insert(R);  
    SetA.insert(Q);  
    SetA.insert(M);  
    set_union(SetC.begin(),SetC.end(),SetB.begin(),SetB.end(),  
              inserter(SetA,SetA.begin()),Q);  
}  
  
void main(void)  
{  
    pthread_t ThreadA,ThreadB;  
    pthread_create(&ThreadA,NULL,threadA,NULL);  
    pthread_create(&ThreadB,NULL,threadB,NULL);  
    pthread_join(ThreadA,NULL);  
    pthread_join(ThreadB,NULL);  
    A = SetA.begin();  
    B = SetB.begin();  
    C = SetC.begin();  
    cout << "Set A contains ";  
    while(A != SetA.end())  
    {  
        cout << *A << " ";  
        A++;  
    }  
    cout << endl << "Set B contains ";  
    while(B != SetB.end())  
    {  
        cout << *B << " ";  
        B++;  
    }  
    cout << endl << "Set C contains ";  
    while(C != SetC.end())  
    {  
        cout << *C << " ";  
        C++;  
    }  
    cout << endl;  
}
```

对于程序清单 5-8 中的程序，要注意几个重要点。当使用多个进程和 IPC 机制时，`threadA()` 和 `threadB()` 对 `SetA`、`SetB` 和 `SetC` 的访问就没有那么容易了。用于实现可以当作进程间通信的集

容器的编码被禁止。请记住，集（set）的概念指不包含副本的集合（collection）。每次插入 SetA、SetB 和 SetC 时需要禁止成员的复制。同时要注意，threadA() 进行一个交集运算，而 threadB() 进行并集运算。交集和并集都是集的主要运算，如果涉及的集很大的话，可能需要较长的处理过程。在操作系统级别维护一个集容器以及它的自然运算是一个开销昂贵的方面。这是多个线程有时胜过多进程的另一个原因。如果需要的共享内存数据结构十分复杂，与使用多个进程和进程间机制相比，使用多个线程将更容易，而且更有效。几乎可以使用共享内存来建立任何类型的复杂数据结构。但读和写多个进程外部的共享内存比读和写位于同一个进程内、且由多个线程共享的数据结构开销更大。

程序清单 5-8 中的程序也突出了被进程内多个函数或线程访问的数据的一个主要问题。即多函数或多线程容易有意或无意覆盖数据。因为 SetA、SetB 和 SetC 的更新不是同步的，所以我们不能肯定地说，在执行 set_intersection() 和 set_union() 运算后，元素被放到这些集中了。而且，在 SetA 中的元素在 threadA() 中被删除。因为 threadA() 和 threadB() 异步执行，SetA 中最后的元素是值得怀疑的。在 threadB() 在 SetA 中插入元素的同时，threadA() 删除 SetA 中的元素吗？还要注意，threadB() 对 SetC 执行 set_union()，而 SetC 保存有 SetA 中的结果。当 threadA() 正删除 SetA 中元素时，这一过程发生吗？这些问题是在多线程处理环境中最大的一个缺陷。这些问题归入数据竞争问题之列。在多线程环境中，同时（simultaneous）访问全局变量和数据结构必须同步化（synchronized）。

四、线程间通信的参数

支持多个线程间通信的另一种形式是参数传递（parameter passing）。在 POSIX、Win32 以及 OS/2 环境中，线程创建 API 都支持线程参数。参数以一种空指针（void pointer）的形式出现。例如，在 POSIX 中，pthread_create 调用可用于创建一个新的执行线程：

```
pthread_create(ThreadId, Null, threadFunction, (void *)X);
```

C++ 中的空指针基本上是一般性指针。空指针可用于指向任何数据类型。所以，就像 char 的指针一样简单，可用 X 的值来传递值，也可能像容器类或其它复杂数据结构的指针一样复杂。当使用线程创建 API 给一个线程传递指针时，这个指针就像 exec 函数家族参数一样使用。exec 函数用于创建子进程，并包含作为形式参数（parameter）传递给子进程的参数（argument）。使用 exec 函数家族与使用线程创建 API 传递参数的区别在于，exec 函数中的参数表示一种单向通信。子进程只复制在 exec 调用中传递的参数值。当线程接收到参数时，它并不是一个拷贝。线程的参数是某些数据位置的地址。对线程中数据的任何修改都将反映在创建该数据的进程中。这与在 exec 调用中子进程所接受的参数相反。子进程对参数所作的任何修改不会反映在父进程中。

程序清单 5-9 中的程序演示了线程参数的使用。函数 main() 中的主线程声明 N 是一个指向有理数对象的指针。创建一个动态分配的有理数对象。main() 函数然后创建两个新控制线程，threadA() 和 threadB()。main() 函数在调用 pthread_create() 函数期间将 N 传递给 threadA() 和 threadB()。虽然 N 是有理数对象的一个指针，但它被 pthread_create() 函数所接受，因为 pthread_create() 中的 void* 参数与任何类型的指针匹配。由 threadA() 和 threadB() 对 N 指针对象的任何修改都将反映在 threadA()、threadB() 和 main() 中。在 threadA()、threadB() 和 main() 中，对 N 的立即可见性可能导

致问题。因为所有的3个线程可能并发执行，我们必须小心不要修改保存在N中的对象，除非修改是同步的。在程序清单5-9的程序中，保存在N中的对象只用在读或复制的场合，因此我们不必担心数据竞争的发生。

程序清单 5-9 演示线程创建过程中参数传递的使用

```
#include <iostream.h>
#include <pthread.h>
#include <rational.h>

rational M(1,2);

void *threadA(void *X)
{
    rational *Q;
    Q = (rational *) X;
    rational Z(3,4);
    M = *Q + Z;
    cout << "thread A" << endl;
    cout << *Q << " + " << Z << " = " << M << endl << endl;
}

void *threadB(void *X)
{
    rational *N;
    rational Q(1,1);
    N = (rational *) X;
    Q = M + *N;
    cout << "thread B" << endl;
    cout << M << " + " << *N << " = " << Q << endl << endl;
}

void main(void)
{
    pthread_t ThreadA;
    pthread_t ThreadB;
    rational *N;
    N = new rational(5,3);
    pthread_create(&ThreadA,NULL,threadA,N);
    pthread_create(&ThreadB,NULL,threadB,N);
    pthread_join(ThreadA,NULL);
    pthread_join(ThreadB,NULL);
    delete N;
}
```

与将全局变量或全局数据结构用作线程间通信机制相比，将参数用作线程间通信的一种形式可能更理想。因为进程中的所有线程都可以访问全局数据，控制哪个线程什么时候做什么困难。通过限制只能访问那些作为参数接受数据的线程，程序员可以控制对数据值和数据结构的访问。通过监视那些作为参数接受数据的线程可以控制同步。当在多线程环境中声明数据为全局时，程序的增强、维护和调试都变得更为困难。

五、文件句柄

当多线程间的共享文件作为一种线程间的通信形式时，同样要小心全局变量。这是因为如果 `threadA()` 移动了文件指针，`threadB()` 也会受到影响。如果文件被 `threadB()` 关闭，而 `threadA()` 试图写入此文件，显然会有冲突。在多线程环境中，序列化或同步化文件访问时也要小心。因为线程可以共享实际的地址、文件读指针、文件写指针、全局变量以及数据结构，所以，必须使用同步和合作技术。第6章探讨了多线程环境中线程间的同步和合作技术。

合作与同步

...通过模块代理，它们每个都按自己的方式简化了。其中有许多相互冲突。这种冲突受控制和制约，而不是解决。

Mind Storms Children, Computers, and Powerful Ideas

——Seymour Papert

任何计算机系统资源都是有限的。内存是有限的。I/O 设备的数量通常受 I/O 端口数量和系统拥有的硬件中断数量的限制。系统中的处理器数量是有限的。执行时间的长短是有限的。正是在这种有限硬件和软件资源的环境中，由多个线程和进程组成的应用要竞争处理器时间、内存位置以及周边设备（peripheral device）。一些线程和进程工作时一起使用系统的有限共享资源来完成一个共同的目标，而另一些线程和进程异步和独立工作，它们竞争相同的共享资源。为了让这些进程和线程都有机会执行和完成工作，操作系统必须决定什么时候由谁使用资源，以及使用多久。请回忆第 4 章所讲的抢占式规划，操作系统可以中断某个线程或进程，而不管它执行的是什么，也不管它处于执行的哪一个阶段。这种抢占是适应竞争系统资源的所有线程和进程必需的。在这种竞争和抢占式的环境中，程序员必须释放多线程应用，希望它们能正确执行。

6.1 竞争条件

在一个应用中，等待程序员尝试使用多线程有几个缺陷。为了其中一些缺陷，让我们看看两个线程，这两个线程用于对某给定系统上所有文本文件进行多个关键字搜索。其中的两个线程为线程 A 和线程 B。线程 A 的任务是搜索系统中的每个目录寻找文本文件。一旦线程 A 发现了文本文件，就将文件的名字放到一个 List 对象中，这个对象线程 A 和线程 B 都可以访问。

List 对象按字母顺序排序。同时，线程 A 发现一个文本文件后，线程 A 增加变量 Count。线程 B 从列表中获取下一个可用的文件名，并搜索文件中用户感兴趣的关键字。一旦线程 B 从列表中读取了文件名，线程 B 就将 Count 减小 1，并从列表中删除这个文件名。线程 A 继续搜索系统

中的所有目录，直到找完了所有的目录为止。线程 B 继续搜索列表中的文件，直到列表中再也没有用来搜索的文件为止。

图 6-1 描述了以上情形中的每个重要组件。线程 A 和线程 B 访问两个共享资源，一个变量 Count 和一个 List 对象。List 对象有序排列。只要添加了文件名，列表就会重新排序，而且只要删除一个文件名，列表也会重新排列。List 对象可能也被调用按要求显示其内容。在这个例子中，线程 A 和线程 B 并发（concurrently）或异步（asynchronously）执行。一旦激活这些线程，可能发生大量问题。线程 B 可能在线程 A 向列表中添加文件名前试图从列表中读取文件名。线程 B 试图减小 Count 的时候，可能线程 A 试图增加 Count。而且当线程 B 正删除一个文件名，或线程 A 添加文件名之时，List 对象可能试图排序自身。线程 B 如何才能知道什么时候不再有需要搜索的文件？列表可能为空，而线程 A 可能仍然搜索文本文件。当线程 B 试图删除线程 A 正试图插入的同一个文件时，将会发生什么呢？如果线程 A 正添加文件名，或线程 B 正删除文件名之时，要求 List 对象显示其内容，结果将会如何？List 对象装满且不能再接受任何文件名后，线程 A 可能还试图给 List 对象添加文件。当 List 对象为空时，线程 B 可能还试图从 List 对象删除某文件名。操作系统决定抢占线程 A 的时候，线程 A 可能正在增加 Count。在线程 A 有机会完成增加 Count 之前，线程 B 可能减小了 Count。

显然，在这种情况下碰到的问题是不可接受的。线程 A 和 B 必须找一种访问 Count、List 对象的合作和同步方式，否则，用户查找多关键字的请求就不会正确完成。在这个例子中碰到的问题都可以称做竞争条件（race condition）。当两个或更多线程或进程试图同时修改同一个共享、可修改数据块时，这种情况称做竞争条件或数据竞争（data race）。

区别共享可修改数据与只读数据是重要的。区别多线程试图修改数据块与多线程只试图读取数据也是重要的。如果多线程试图同时访问一个不能修改的数据块（即只读内存或常量对象），就不存在数据竞争的问题。同样，如果多线程只是试图同时读取数据块，数据竞争也不会发生。为了让竞争条件存在，目标内存块必须是可修改的，而且线程必须试图同时访问这个块，至少其中一个线程试图修改这个内存块。

在前一个例子中，变量 Count 和 List 对象表示共享可修改内存块。当线程 A 试图增加 Count，而线程 B 试图减小 Count 时，此时就创建了竞争条件。当线程 B 试图从 List 对象中删除线程 A 正试图添加的同一个文件时，也创建了竞争条件。当 List 对象试图排序，而线程 A 正添加一个文件名或线程 B 正删除一个文件名，此时创建了另一个竞争条件。创建竞争条件是多线程或并发编程的主要缺陷之一。

多线程编程的第二个主要敌人是死锁（deadlock）。让我们使用图 6-1 中的 4 个组件线程 A、线程 B、Count 变量以及 List 对象来说明死锁是如何产生的。我们可能试图允许每个线程轮流排它性访问 Count 变量和 List 对象，以解释线程 A 和线程 B 之间可能的竞争条件。可以提供一种 lock() 机制来实现。lock() 机制使锁定的变量只能被首先调用 lock() 机制的线程访问。假如线程 A 刚刚定位了一个文本文件，同时需要增加 Count 的值并将文件名添加到 List 对象。如果线程 A 想避免创建一项数据竞争，它首先要锁定 Count 变量，然后锁定 List 对象。线程 A 完成锁定后，它将增加 Count 的值，并且将新文件名添加到 List 对象。这一方案似乎很简单，其实存在一些问题。线程 B 可能试图以线程 A 的同样方式执行锁定。不过，线程 B 可能选择首先锁定 List 对象。线程 A

成功锁定 Count 变量，线程 B 成功锁定 List 对象，线程 A 可能试图锁定线程 B 已经锁定的 List 对象，同时线程 B 可能试图锁定线程 A 已经锁定的 Count 变量。图 6-2 演示了这种矛盾的状态。

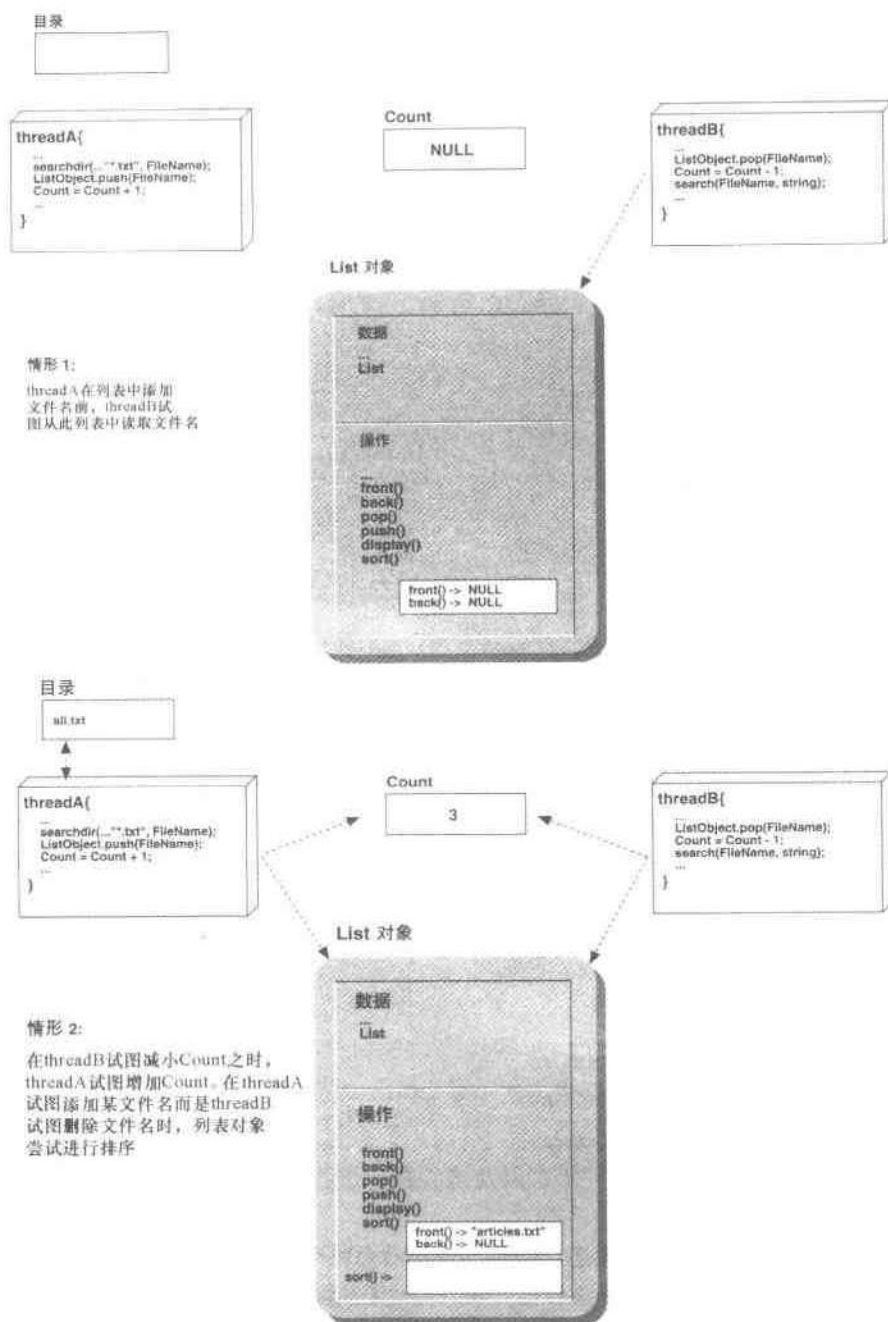


图 6-1 演示共享两个资源的两个线程。这两个资源是变量 Count 和 List 对象

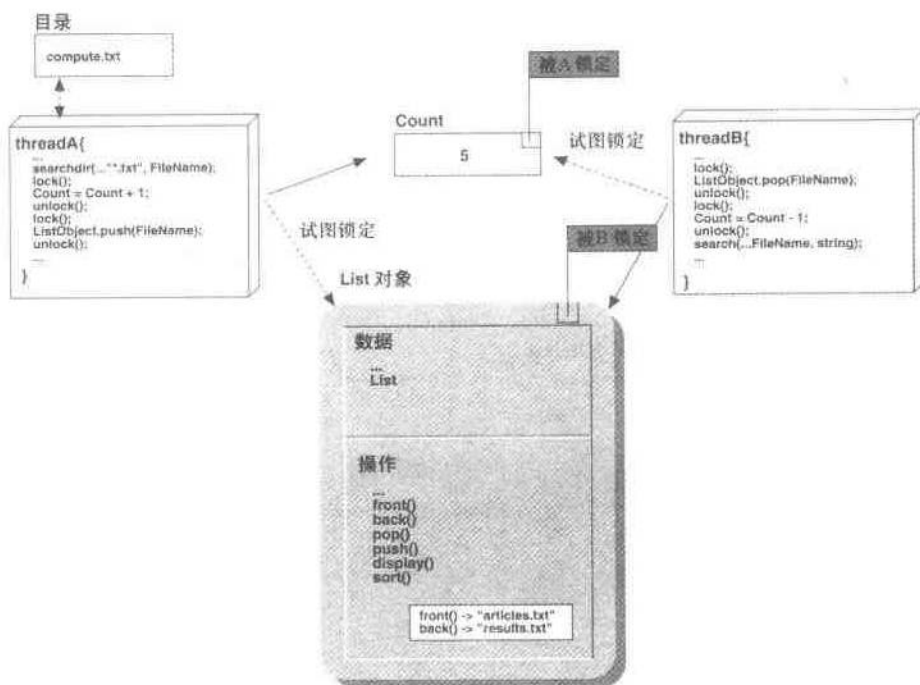


图 6-2 演示线程 A 如何锁定 Count 资源，同时线程 B 锁定 List 对象资源。

线程 A 不能锁定 List 对象，线程 B 也不能锁定 Count 变量。这称做死锁

由于线程 A 要等待机会锁定 List 对象，所以它不能继续执行。它不能锁定 List 对象，是因为线程 B 已经锁定了 List 对象。线程 B 由于等待机会锁定 Count 变量，所以也不能继续执行。它不能锁定 Count 变量，是因为线程 A 已经锁定了 Count 变量。线程 A 等待着线程 B，而线程 B 同时等待着线程 A。这种情形就称做死锁（deadlock）。当锁定、挂起或冻结多个线程时，相互间等待对方释放某些资源，此时存在死锁。竞争条件和死锁大概是多线程处理应用的最大缺陷。多个线程和多个任务必须同步化阻止竞争条件。我们必须找到一种让多个线程和多个任务合作使用资源避免死锁的途径。

实质上有 3 类主要的同步：

- 数据同步（Data Synchronization）；
- 设备同步（Device Synchronization）；
- 任务同步（Task Synchronization）。

表 6-1 列出了 3 种主要类型的同步，以及多线程或进程并发使用时，同步化需要的常见元素。

同步允许多个线程或进程同时激活，同时共享资源而不干扰对方的操作。同步进程临时序列化多个线程和任务的执行，防止竞争条件或死锁。序列化仅在必须对硬件或软件资源按“一次一个”（one-at-a-time）的方式访问时才发生。请注意，太多的序列化会削弱多线程的优势。

表 6-1 3 种主要类型的同步及其描述

同步类型	描 述
数据同步	防止竞争条件时必需。它允许并发线程和进程安全访问内存块。访问全局变量、共享内存和文件必须同步化
硬件同步	若干硬件设备需要执行一项任务或一组任务时必需硬件同步。它要求进程间的通信、对实时操作和优先权的严格控制
任务同步	防止竞争条件时必需。它强加逻辑进程的前置条件和后置条件

6.1.1 数据同步

数据同步是阻止竞争条件必需的，而且允许并发线程或进程按一种安全模式使用内存块。数据同步控制何时可以修改数据块。在多线程环境中，一般情况下，作为共享内存、全局变量以及文件来访问这种数据必须同步化。

6.1.2 硬件同步

当若干硬件设备一起使用来执行某个任务或一组任务时，此时需要硬件同步。例如，图 6-3 中的两个线程都访问 MCI（多媒体设备）。线程 A 访问一个波表设备，线程 B 访问一个视频播放器。线程 A 中的声音必须与线程 B 中的场景对应。两种设备都有不同的定时、内存需求以及反应时间。



图 6-3 演示线程 A 和线程 B 如何访问 MCI 设备

为了准确混合场景和声音，同时使用这些设备的进程或线程必须同步化。这种同步需要进程间的通信，同时严格控制实时操作和优先权设置。试图协调使用设备（如 MCI 多媒体设备）的这些线程必须紧紧耦合（couple）。一个线程挂起，而另一个线程继续执行，这不是所期望的。例如，

我们不希望图 6-3 中的视频被操作系统抢占，而同时，应当与视频匹配的声音还在继续播放。这是不能接受的。

6.1.3 任务同步

逻辑任务也必须同步化。这与同步数据或同步硬件设备有些不同。对于数据同步，目标是阻止竞争条件。硬件同步允许多个设备联合使用来完成一个目标或多个目标。硬件同步也可以用于让多个线程或进程共享单个硬件设备。当硬件同步允许共享单个硬件设备时，其目标就是阻止线程或进程破坏设备的状态。任务同步加强逻辑进程的前置条件（precondition）与后置条件（postcondition）。例如，假如有 3 个并发工作的线程，其目标是显示一个列表中的项，列表要被排序和搜索，显示发现的项。于是，我们首先要排序这个列表。可以允许对列表并发进行多个搜索，然后希望显示搜索的所有结果。如果这些线程没有正确协调，负责显示结果的线程可能在搜索到结果前试图显示它们。这就违背了进程的后置条件。这种情形的先决条件是列表在搜索前必须已经排序。如果搜索在列表排序前开始，搜索结果可能是错误的。请注意，有时任务同步只是数据同步的一个特例。这些线程的流程控制如图 6-4 所示。

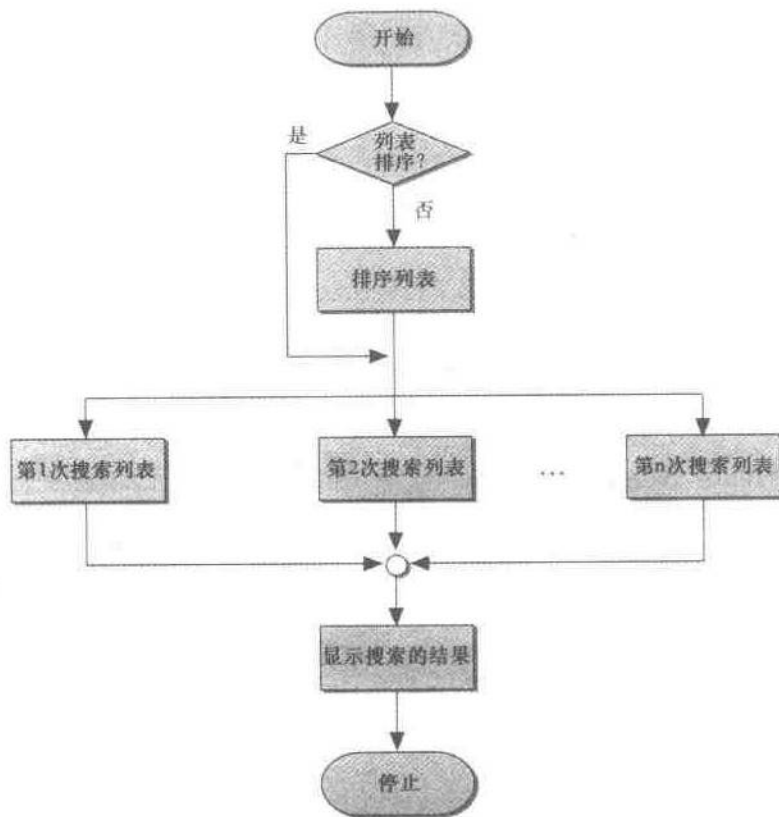


图 6-4 描述排序、多重搜索以及显示线程的控制流程

排序线程必须首先发生。多重搜索允许在排序后发生，最终这些搜索加入到一个显示搜索结果的线程中。

6.2 同步关系

现在，我们已经明白了在多个线程或多个进程环境中，需要同步化的3个基本类别。我们可以定义存在于两个线程间的同步关系。在一个进程中，任何两个线程之间有4种基本同步关系。这4种基本同步关系及其描述如表6-2所示。

表 6-2 4 种基本同步关系及其描述

基本同步关系	描 述
SS (start-start)	线程 B 启动后线程 A 才能启动。线程 B 可以在线程 A 启动的同时或之后启动，但决不能在其之前启动
FS (finish-to-start)	线程 A 完成一定的任务后，线程 B 才能启动
SF (start-to-finish)	线程 B 完成一定的任务后，线程 A 才能启动
FF (finish-to-finish)	线程 B 完成后，线程 A 才能完成。线程 A 可以在线程 B 完成后完成，但不允许线程 A 在其之前完成

如果有两个线程 A 和 B，那么第一种关系称做 SS (start-start) 关系。也就是说，线程 A 等到线程 B 开始后才能开始。线程 B 可以在线程 A 开始的同时开始，也可以在线程 A 开始之后开始，但决不能在线程 A 开始之前开始。图 6-3 中的线程就是这种关系。声音序列必须等待视频序列。在影像开始前，不应该听到声音。因为定时条件，有时允许视频在声音之前开始，但是反之则是不允许的。

下一个关系是 FS (finish-to-start) 同步。也就是说，进程 B 一直等到进程 A 结束或完成一定操作后才能开始。例如，如果进程 B 正从一个与进程 A 相关的管道读取数据。在进程 B 读取管道前，进程 A 需要至少写入管道一个元素。在这种情况下，进程 B 开始操作前，进程 A 需要结束一个操作。同样，如果进程 B 需要对进程 A 必须排序的列表执行二分检索，进程 B 必须与进程 A 同步化，让进程 B 等到进程 A 结束排序后才能开始检索。在以上两种场合中，如果进程 B 试图执行操作前，进程 A 没有完成它的操作，进程 B 将失败或得到不正确的结果。FS 关系通常暗示着一种信息依赖性。在信息依赖性中，一个线程需要另一个线程的 ITC 来正确操作。下一个关系与 FS 相反。SF (start-to-finish) 同步关系表示线程 A 等到线程 B 开始后才能结束。这种关系常见于以下情形：父进程需要来自子进程的 IPC 来完成，或进程或线程递归调用自身的情形。在 SF 关系中，子线程或进程为父进程提供了信息或所需事件后，它才能继续执行。

最后是 FF (finish-to-finish) 同步关系，它表示线程 A 一直等到线程 B 结束后才能结束。线程 A 可以在线程 B 结束后结束，但线程 A 不允许在线程 B 结束前结束。以程序清单 6-1 中的程序为例，其中主线程 A 创建了两个线程 B 和 C。线程 A 动态分配有理数对象 X，并将它传递给线

程 B 和线程 C。我们不希望线程 A 在线程 B 和线程 C 结束前结束，因为线程 A 破坏了有理数对象 X 的指针。如果线程 B 和线程 C 开始使用 X 时，它已不再得到分配，那么就可能发生某类片断错误或数据访问违规。为了阻止这种错误的发生，我们使用 `pthread-join()` 函数同步化线程 A 与线程 B 和线程 C 的终止。这个函数导致线程 A 在线程 B 和线程 C 结束前不会结束，因此创建了一种 FF 同步。

滞后时间

有时，同步关系需要针对特定定时信息加以扩充。这意味着设计同步关系时，必须考虑时间和事件。例如，如果线程 A 和 B 并发执行，其中线程 A 执行一个通信任务，线程 B 是一个超时监视器。线程 A 和线程 B 可以通过 SS 关系同步化。在线程 A 启动通信任务前，线程 B 没有必要检查超时条件。不过，一旦线程 A 启动了它的通信任务，而且持续数毫秒没有动作，线程 B 将发出超时消息。在这种情况下，线程 B 在发出超时消息前使用的是滞后时间 (lag time)。滞后时间用于进一步定义同步关系。滞后时间需要给同步关系的定义加上一个时间元素。例如，我们可以说线程 A 和 B 是一种 SS 同步，另外还要求线程 B 必须等待线程 A 启动 10 纳秒后才能启动。

6.3 进程同步机制

为了有效地处理多个线程或进程间的竞争条件，有几种同步机制是有用的。这些机制为：

- 信号量 (semaphore)；
- 条件变量 (conditional variable)；
- 临界区 (critical section)。

我们可以结合 SS、FS、SF 和 FF 同步关系与同步机制来消除在多线程编程中发生竞争条件和死锁的可能性。尽管以上同步关系和机制不能完全防止所有类型的竞争条件和死锁，但它们形成了在 C++ 编程中实现并发的一个坚实基础。

6.3.1 信号量提供钥匙

信号量是一种特殊的变量。E. W. Dijkstra 在 *Cooperating Sequential Processes in Programming Languages* 一书中介绍了信号量 (semaphore) 的基本概念及其操作。信号量是一个受保护的变量，它仅能被非常具体的操作来访问。信号量用于帮助线程或进程同步访问一些共享、可修改内存块，或者控制对一些设备的访问。信号量充当共享资源的一种钥匙。在某个时刻，这把钥匙只能由一个线程或进程拥有。无论谁拥有了这把钥匙，它就可以锁定资源进行排它性地使用，在允许资源的进一步访问之前，其它进程或线程必须等待资源被释放。具有此信号量的线程就被认为占有此信号量。任何试图占有这个信号量的其它线程或进程一般都被阻塞，一直要等到信号量的占有者释放它为止。然后队列中的下一个进程 (由实际使用的规划策略决定) 可以接过信号量的占有权。一旦获得了信号量的占有权，信号量保护的资源就可以被访问了。

对信号量的操作 (P、V、Lock 和 Unlock)

因为信号量是一个受保护和封装的变量，只有一定的操作才允许操作信号量。对操作的限制

可应用于所有线程，甚至应用于信号量的占有者。信号量有若干类，每类信号量都有一套特定的可允许操作。一种信号量类型是二进制信号量 (binary semaphore)，它的值只能是 0 或 1。对于二进制信号量只能进行两种操作。按习惯，这两种操作标记为 P() 和 V()。P() 操作基本上是一个减量操作。如果 Mutex 为信号量，那么 P(Mutex) 的逻辑实现如下：

```
if (Mutex > 0) {
    Mutex--;
}
else {
    Block on Mutex;
}
```

对应的增量操作是 V(Mutex)，逻辑实现如下所示：

```
if (Blocked on Mutex N processes) {
    pass Mutex on;
}
else {
    Mutex++;
}
```

我们在这里使用“逻辑实现” (logically implement) 一词，是因为实现信号量操作的方式与系统有关，也许与逻辑实现看起来不相像。不过，操作的意义是一样的。另一种信号量是计数信号量 (counting semaphore)。计数信号量保证只以非负整数值出现。计数信号量也受到 P() 和 V() 操作的限制。对于信号量操作，我们在这里使用的是 P 和 V，其它名字也常常见到。例如，还可以发现 lock()、unlock()、signal()、wait()、down() 和 up() 等名字用来指信号量操作。本书中我们使用 lock() 和 unlock() 用来指信号量操作。

原子性 (atomicity) 和不可分割性 (indivisibility)

信号量操作不是可以作用于普通整数值的一般赋值、增量和减量操作。信号量操作确保是不可分割的。一旦启动了信号量操作，它就不能由于任何原因被操作系统抢占。这一点很重要，因为一条 C++ 语句可能产生许多机器级指令。例如：

```
Z = X + Y;
```

可以分解为多条机器级指令：

```
mov    ax, X
mov    bx, Y
add    bx, ax
mov    Z, cx
```

```
...
...
```

有人认为一旦计算机启动了一条 C++ 语句的执行，它就会完成，但通常并非如此。在前一个例子中，在 X+Y 的结果进入变量 Z 之前，操作系统可以在任何语句点抢占该进程。所以，甚至简单的 C++ 测试和设置结合语句也可能具有欺骗性。如下面语句：

```
if (Mutex > 1)
    lock;
```

似乎是一条简单的指令，一旦启动后就完成，而不会被中断。情况可能并不是这样。C++ 的

这一条语句实际上分解成若干机器级指令，它们可以被处理器或操作系统打断。

```
mov ax, Mutex
cmp ax, 1
jle lock
...
...
```

像这样，语句中可能发生的在机器级抢占在多线程程序中是危险的，原因有以下几条。主要原因是在这种语句执行期间的抢占可能导致竞争条件。如果存在线程 A 和 B，它们都试图执行这一条语句，在执行简单 if-then-lock 语句所需的机器级指令期间，我们就不能保证线程 A，或者线程 B，或者两者都不被抢占。当线程 A 运行到 mov 语句时，可能被抢占，当线程 B 运行到 cmp 语句时，也可能被抢占，致使尝试同一进程的其它线程被锁定。此时，对信号量的操作就要发挥作用了。信号量操作是原子性（atomicity）或不可分割性（indivisibility）的操作。一旦开始了信号量操作，它就能保证在不在被抢占的情况下运行到完成。可以按多种途径来完成，而且总是与系统相关的。例如，一旦锁定了信号量，通过确保临时取消中断，可以确保对信号量的操作。要记住的重要一点是，程序员不能用普通布尔变量和布尔操作取代特殊的信号量变量和信号量操作，因为大部分普通操作不是原子性的，它可能被中断。虽然 C++ 语言与某些汇编语言级指令有一对一关系，但大部分 C++ 语句均需要多条机器级指令。在这些机器级指令中，就可能发生抢占。

6.3.2 信号量类型

表 6-3 列出了常见的 4 种信号量类型、它们的操作以及基本用途。

表 6-3 常见的 4 种信号量类型、它们的操作以及基本用途

信号量类型	基本用途与操作
互斥信号量	帮助在代码临界区（初始化、锁定请求、try 块、取消锁定、析构）实现互斥的机制
事件互斥量和条件变量	用于支持线程间的广播机制。它允许一个线程向其它线程广播通知某事件的发生。当线程锁定某事件互斥量时，它将阻塞到接收到广播（等待、延迟、销毁）为止
等待多个条件	与事件互斥量相同，但它扩展到包括多个事件或条件（等待、延迟、销毁）

习惯上，将信号量称做二进制信号量、计数信号量，或者一般信号量，但在 NT 和 OS/2 操作系统中，对于以上同样的概念，称做互斥信号量（mutex semaphore）、事件信号量（event semaphore）和互斥等待信号量（mutex wait semaphore）。名字尽管不同，但功能性是一样的。对于本书所讨论的每种环境（POSIX Pthread、OS/2 以及 NT），互斥量的实现有些不同，但这些机制的用途是相同的：防止竞争条件与死锁。

互斥信号量

在线程的代码中，共享数据块被修改，或者共享资源被更改之处称做临界区（critical section）。这个术语是恰当的，因为正是在临界区创建竞争条件和死锁。如果正确管理多线程程序的临界区，就可以避免使用并发技术时可能发生的大部分缺陷。在多线程编程中的第一道防线是保护程序中

的临界区，将它们锁定，不让其它线程或进程访问。这种保护临界区，只允许一个线程或进程在某一时刻访问它的过程称做互斥（mutual exclusion）。帮助实现互斥的机制通常称做互斥量（mutex）。POSIX 线程标准有一个互斥类型，名叫 `pthread_mutex_t`，它用于给临界区提供基本的访问保护。与基本互斥量一起，NT 和 OS/2 环境还支持事件互斥量和多重等待互斥量。

互斥信号量是经典二进制信号量的一个增强版本。从其最单纯的形式上讲，它是一个不存在特定实现细节的理论概念。另一方面，互斥信号量包含实际可行信号量机制的所有必需操作。表 6-4 列出了可以操作互斥信号量的 5 种基本操作。

表 6-4

可以操作互斥信号量的 5 种基本操作

对互斥量的操作	描 述
初始化	初始化进程将分配保存信号量所需要的内存，并赋予内存初始值。它还决定信号量是否被占有、非占有、私有或共享
锁定请求	互斥量可以被锁定或取消锁定。当锁定时，互斥量导致获得锁定的线程具有该互斥量的占有权。如果多个线程试图同时访问该互斥量，除了授予锁定的线程外均被阻塞
Try 锁定	用于测试互斥量，检查它是否已被占有。使用它时，将检测互斥量的占有权。如果互斥量被占有，但没有阻塞线程，则函数返回一个错误。如果互斥量没有被占有，则锁定成功。如果可以等待互斥量一段时间。如果在这段时间内，它没有取消锁定，则线程继续执行
取消锁定	互斥量释放或取消锁定请求将导致其它线程等待该互斥量的取消锁定。当其中一个线程获得了互斥量的占有权时，所有剩下的线程都仍然等待访问权的再次阻塞
析构	释放与互斥量相关的内存。如果互斥量被占有或某线程仍然等待着该互斥量，可以销毁或关闭内存

互斥信号量初始化

互斥信号量需要初始化。大部分互斥信号量为共享内存变量。也就是说，它们在创建特定用于信号量的内存块。互斥信号量的初始化过程将分配保存信号量需要的内存，并且赋予内存初始值。部分初始化过程还决定信号量是否被占有或不占有，私有或共享。当初始化一个互斥信号量为共享时，允许该互斥信号量被不同进程中的线程以及同一进程中的线程共享。表 6-5 显示了在创建或初始化互斥信号量时可以设置的互斥信号量属性。

表 6-5

创建或初始化互斥信号量时可以设置的互斥信号量属性

互斥信号量可设置属性	描 述
被占有或未被占有	指定设置了信号量初始状态的一个标志。如果为被占有，请求该信号量的其它所有线程都将阻塞。一旦操作系统创建了它，调用线程将立即占有该信号量，同时可以访问此资源。如果它没有被任何其它线程占有，则请求此信号量任一线程马上可以获得占有权

续表

互斥信号量可设置属性	描 述
命名或匿名	命名信号量总是被共享。知道它的名字的所有进程都可以使用。当应用程序在创建信号量的函数中指定一个名字时，操作系统将为信号量创建一个名字。如果没有指定名字，则创建一个匿名信号量。匿名信号量可以为私有或共享
私有或共享	私有信号量未命名，而共享信号量已命名，也可以未命名。私有信号量通过它们的句柄来识别，只能被进程内的线程使用。命名共享信号量使用名字或句柄打开。它们可以被多进程中的线程使用

同步变量不会像常规变量一样保证得到初始化。如果我们声明 `int X=1`，就可以认为 `X` 已经初始化了。不过，当初始化互斥信号量时必须小心，因为初始化不是一个得到保证的操作。这意味着初始化过程可能失败。互斥量初始化失败的原因有多种。系统可能将所有的可用空间都用于信号量。命名信号量可能已经存在，或者发生了其它内存分配问题。因此，在锁定、取消锁定或销毁操作中使用任何信号量之前，确保信号量最初正确初始化了。互斥量初始化通常由某种类型的函数调用来完成。

```
//POSIX 互斥量初始化
pthread_mutex_t MHandle
pthread_mutex_init(&MHandle, NULL);

//OS2 互斥量初始化
HMTX MHandle
DosCreateMutexsem(NULL, &MHandle, 0, 0);
```

这些函数创建互斥信号量。`pthread_mutex_init()`函数是一个在 POSIX Pthreads 包中如何创建和初始化互斥量的例子。Null 参数允许线程具有缺省属性。`DosCreateMutexsem()`函数是一个在 OS/2 环境中如何创建和初始化互斥量的例子。函数的名字不同，但功能是相同的。

互斥信号量锁定请求

正确初始化互斥量之后，互斥量可以锁定、取消锁定，或者销毁。锁定互斥量的过程导致获得锁定的线程拥有互斥量的占有权。尽管许多线程可能同时试图获取互斥量持占有权，但一次只能让一个线程占有互斥量。当多个线程试图获取互斥量访问权时，除了一个线程外，其它线程都将阻塞此互斥量调用。程序清单 6-1 中的程序使用 POSIX 线程库演示多线程和互斥量的使用。这个程序有两个线程 `testThread()`和 `testThread2()`。每个线程输出它的线程 ID 10 次。不过，只有线程获得了互斥量的占有权时才允许输出它的线程 ID。

程序清单 6-1 当线程 `testThread()`和 `testThread2()`获得了互斥量的占有权时，它们输出各自的线程 ID。每个线程输出为 10 次

```
#include <iostream.h>
#include <pthread.h>
#include <stdlib.h>
```

```
pthread_mutex_t MyLock;

void *testThread(void *X)
{
    int N;
    for(N = 0; N < 10; N++){
        pthread_mutex_lock(&MyLock);
        cout << pthread_self() << " I'm Alive thread 1" << endl;
        pthread_mutex_unlock(&MyLock);
        pthread_yield();
    }
}

void *testThread2(void *X)
{
    int N;
    for(N = 0; N < 10; N++){
        pthread_mutex_lock(&MyLock);
        cout << pthread_self() << " I'm Alive thread 2 " << endl;
        pthread_mutex_unlock(&MyLock);
        pthread_yield();
    }
}

void main(void)
{
    pthread_init();
    pthread_t ThreadId;
    pthread_t ThreadId2;
    if(pthread_mutex_init(&MyLock, NULL)){
        cerr << "could not initialize mutex " << endl;
        exit(1);
    }
    pthread_create(&ThreadId, NULL, testThread, NULL);
    pthread_create(&ThreadId2, NULL, testThread2, NULL);
    pthread_join(ThreadId, NULL);
    pthread_join(ThreadId2, NULL);
}
```

其中有一个叫 `MyLock` 的互斥量。`MyLock` 的占有权通过对 `pthread_mutex_lock()` 的调用来获取。因为两个线程都调用 `pthread_mutex_lock()`，一个线程获得互斥量的占有权，另一个线程调用函数时被阻塞。也就是说，不允许被阻塞的线程继续完成函数调用，直到互斥量的占有权通过调用 `pthread_mutex_unlock()` 释放为止。在程序清单 6-1 的程序中，只有两个线程竞争 `MyLock` 的占有权。所以，当 `MyLock` 的占有权被释放时，一直等待的线程将立即获得互斥量。不过，当存在多个线程等待着一个互斥量时，情况并非如此。当阻塞了多个线程，而且等待着互斥量的时候，

必须基于优先权和使用的规划方案来决定由哪一个线程占有互斥量。如果线程的优先权不相等，则通常最高优先权的线程第一个获得释放后互斥量的占有权。如果所有的线程优先权相等，则规划器可能使用一种轮询或 FIFO 途径来让线程获得互斥量的占有权。谁获得释放后的互斥量由规划器和线程的属性来决定。

程序清单 6-1 中的程序使用互斥量来保护对 cout 对象的访问。因为两个线程都从进程继承了标准输入和标准输出，所以，两个线程都对 cout 对象具有相等的访问权。因为 cout 对象被两个线程共享，并被两个线程修改，所以，cout 对象被看作处于一个临界区。互斥量 MyLock 用于保护 cout 对象的状态免于被破坏。如果 cout 对象没有受到保护，那么 testThread() 的输出可能不可预测地与 testThread2() 的输出混合在一起。互斥量 MyLock 是一种最简单类型的同步机制。它的工作是充当一个交通警察，阻塞其它线程，而让某个线程继续，直到适当的时间为止。

6.3.3 自愿互斥量策略

程序清单 6-1 中程序里的 testThread() 和 testThread2() 在试图获得对 cout 对象的访问权之前，必须通过自愿调用 pthread_mutex_lock() 来合作。任何一个线程在可以使用 cout 对象之前，不必首先调用 pthread_mutex_lock()。如果这样的话，cout 的状态就会受到破坏。互斥量策略是自愿的。也就是说，没有强行的规则要求临界区的线程使用互斥量。使用互斥量访问同步化是一种编程策略，与句法上的强制实施相反，它是根据实际来实施的。自愿使用互斥量来同步可能是多线程环境中主要缺陷的产生原因。如果 20 或 30 个线程具有同一个临界区，但不是全体一致地保护这些临界区，就可以导致一些值得怀疑的结果或奇怪的程序崩溃。如果程序相当大，访问临界区的代码可能隐藏于其它复杂的逻辑代码中。程序员可能没有意识到未保护临界区正被访问。这个问题可以在 C++ 中通过封装来解决。第 8 章将探讨简化互斥量使用的几种方法。

6.3.4 使用互斥量锁定防止竞争条件

一旦标识了所有的临界区，可以在临界区周围放置互斥量来解决。程序清单 6-2 中的程序有两个临界区。它有一个队列叫 TextFiles 以及对象 iostream cout。两个对象都可以被程序中的所有线程访问。两个对象被程序中的线程共享，而且两个对象被程序中的两个线程修改。被多个线程共享和修改创建了常常导致数据竞争的条件。

程序清单 6-2 这个程序使用了两个线程。第一个线程查找包含字符串 txt 的所有文件名。第二个线程通过指定的关键字列表查找这些文件。程序使用互斥量来保护临界区。

```

1  #include <dirent.h>
2  #include <set.h>
3  #include <cstring.h>
4  #include <fstream.h>
5  #include <algo.h>
6  #include <string.h>
7  #include <list.h>
8  #include <stack.h>
9  #include <stddef.h>

```

```
10  #include <process.h>
11  #define INCL_DOSPROCESS
12  #define INCL_DOSSEMAPHORES
13  #include <os2.h>
14
15  queue< list<string> > TextFiles;
16  set<string,less<string> > KeyWords;
17  set<string,less<string> > SearchWords;
18  void threadA(void *X);
19  HMTX SemHandle;
20  string InFile;
21  string OutFile;
22
23
24
25  void threadA(void *X)
26  {
27      string Temp;
28      string FileName;
29      less<string> Comp;
30      ifstream In;
31      ofstream Out;
32      In.open(InFile.c_str());
33      Out.open(OutFile.c_str());
34      while(!In.eof())
35      {
36          In >> Temp;
37          if(!In.eof()){
38              KeyWords.insert(Temp);
39          }
40      }
41      In.close();
42      while(TextFiles.empty())
43      {
44
45      }
46      while(!TextFiles.empty())
47      {
48          DosRequestMutexSem(SemHandle,5000);
49          FileName = TextFiles.front();
50          TextFiles.pop();
51          DosReleaseMutexSem(SemHandle);
52          In.open(FileName.c_str());
53          SearchWords.erase(SearchWords.begin(),
54                             SearchWords.end());
54          while(!In.eof() && In.good())
```



```
55     {
56         In >> Temp;
57         SearchWords.insert(Temp);
58     }
59     In.close();
60     if(includes(SearchWords.begin(),SearchWords.end(),
61         Keywords.begin(),Keywords.end(),Comp)){
62         DosRequestMutexSem(SemHandle,5000);
63         cout << "Thread A match found in " << FileName << endl;
64         Out << FileName << endl;
65         DosReleaseMutexSem(SemHandle);
66     }
67 }
68
69 Out.close();
70 }
71
72
73 void main(int argc, char *argv[])
74 {
75
76     DIR *DirP;
77     unsigned long Result = 0;
78     struct dirent *EntryP;
79     DirP = opendir(argv[1]);
80     InFile = argv[2];
81     OutFile = argv[3];
82     if(argc == 4){
83         string Temp;
84         DosCreateMutexSem(NULL,&SemHandle,0,0);
85         Result = _beginthread(threadA,8192,NULL);
86         if(DirP == NULL){
87             cerr << " Could Not Open " << argv[1] << endl;
88             exit(1);
89         }
90         do{
91             EntryP = readdir(DirP);
92             if(EntryP){
93                 Temp = EntryP->d_name;
94                 if(Temp.contains(".TXT")){
95                     DosRequestMutexSem(SemHandle,5000);
96                     cout << "Found " << EntryP->d_name <<
97                         " in Main Thread " << endl;
98                     TextFiles.push(EntryP->d_name);
99                     DosReleaseMutexSem(SemHandle);
100                 }
101             }
102         }while(EntryP != NULL);
103     }
```

```

100     }
101     }while(EntryP);
102     closedir(DirP);
103     DosWaitThread(&Result,DCWW_WAIT);
104     DosCloseMutexSem(SemHandle);
105     }
106
107 }
108
109
110

```

程序清单 6-2 中有两个线程：一个主线程和一个次线程，叫 `threadA()`。主线程搜索当前目录查找包含特定字符串的文件名。如果找到了包含特定字符串的文件，就将文件放在一个名叫 `TextFiles` 的 STL 队列对象中。当它根据特定字符串搜索文件名时，它将文件名插入 `cout` 对象中，以便在标准输出上显示。第二个线程是 `threadA()`，它一次删除一个文件，搜索文件中的一套关键字，用户已经将这套关键字放在一个名叫 `query.txt` 的文件中。如果 `threadA()` 找到了一个包含所有关键字的文件，则 `threadA()` 将文件名插入 `cout` 对象，以便在标准输出显示。主线程和 `threadA()` 并发执行。两个线程都能修改队列对象 `TextFiles` 和 `iostream` 对象 `cout`。主线程插入一个文件名的同时，可能 `threadA()` 正试图删除同一个文件名。这种情况自然会构成一个竞争条件。

为了防止竞争条件，程序清单 6-2 中的程序在主线程和 `threadA()` 中使用互斥量来保护临界区。在主线程试图将文件名插入到 `TextFiles` 队列中，或者给 `cout` 对象发送消息之前，主线程锁定互斥量。此时，`threadA()` 不能修改 `TextFiles` 队列或 `cout` 对象。一旦主线程对队列和 `cout` 对象作出了修改，主线程取消锁定互斥量，让 `threadA()` 获得互斥量的占有权。另一方面，如果 `threadA()` 首先锁定了互斥量，那么主线程将被一直阻塞到 `threadA()` 释放互斥量为止。

当程序试图防止竞争条件时，这种处理过程还有一个问题。如果 `threadA()` 在主线程给队列对象添加文件名之前首先得到互斥量，`threadA()` 可能试图从空队列中删除一个文件名。为阻止这种情况的发生，添加如下语句：

```

while(TextFiles.empty())
{
    }
}

```

这条语句可以让 `threadA()` 处于一个繁忙的循环中，直到 `TextFiles` 队列中至少存在一个文件名为止。`threadA()` 现在可以首先执行，而不会从一个空队列中删除文件了。不过，这个繁忙循环不是我们所期望的。首先，它访问互斥量外部队列对象的一个成员函数。其次，它不必要地过多占用处理器。如果程序在一个拥有多个处理器的机器上执行，这种密集、繁忙的循环将导致 `threadA()` 独占处理器，于是影响了处于就绪队列中的其它线程的吞吐量。更好的办法是，在队列对象中不存在文件之前，一直阻塞 `threadA()`。以后我们会看到，使用事件信号量或条件变量也可以解决这种问题。

一、互斥信号量 `try_lock` 请求

通常情况下，对于当前正被一个线程占有的互斥量，多线程程序的逻辑应当让另一个线程在

互斥量上阻塞。在某些情况下必须测试互斥量，而且如果它已经被占有，可能还要执行另外的动作。例如，为了防止线程在已被占有的互斥量上一直阻塞，我们可能想要调用 `try_lock()` 互斥量函数。当线程使用 `try_lock()` 函数时，就测试了该互斥量的占有权。如果它已经被占有，`try_lock()` 函数返回一个错误，但不会阻塞线程。如果互斥量没有被占有，则锁定成功。有时，通过让 `mutex_lock()` 接受一个指定了计数前等待多长时间参数来实现 `try_lock()` 函数。指定一个值后，`mutex_lock()` 函数就不会无休止地等待锁定的获得了，但该值的生存期过后，它还会继续等待。例如：

```
DosRequestMutexSem(SemHandle, 5000);
```

这个函数要求被请求的互斥量占有权生存期不超过 5 秒。如果在 5 秒的生存期过期之前获得了占有权，函数可能在函数调用后继续。如果在 5 秒的生存期过期时，还没有获得占有权，`DosRequestMutexSem` 返回一个错误代码，并且允许函数在互斥量请求后继续。在 POSIX 线程环境中，`pthread_mutex_trylock()` 实现同样的结果。如果互斥量不可用，`pthread_mutex_trylock()` 函数返回一个错误代码，而且允许函数在不被阻塞的情况下继续。

互斥量通常充当计数或资源信号量。一个给定的互斥量可能只接受有限的锁定请求。例如，在 OS/2 环境中，对于一个信号量只允许 65 535 个锁定请求。多于 65 535 的任何请求都将返回一个错误，而且线程不会阻塞。每作出一个互斥量锁定的请求，就将计数器增加 1。每作出一个互斥量释放的请求，就将计数器减小 1。

二、互斥信号量取消锁定 (unlock) 请求

互斥量取消锁定和释放请求会导致等待该互斥量的其它线程取消锁定。一旦线程获得了互斥量的占有权，仍然希望访问互斥量的所有剩余线程再次被阻塞。只有互斥量的占有者可以取消锁定或释放互斥量。大部分线程实现并不允许非占有者释放、取消锁定或关闭互斥量。当取消锁定互斥量后，若同时还有其它线程在该互斥量上阻塞，则具有最高优先权的线程通常交付这个互斥量。如果所有的阻塞线程都具有相同的优先权，线程可以按 FIFO 方式获取可用互斥量的访问权。

三、互斥信号量析构

释放 (release) 或取消锁定 (unlock) 互斥量不会释放 (free) 与互斥量关联的内存。释放 (release) 只是放弃互斥量的占有权，使其它占有者可以锁定它。为了释放 (free) 与互斥量关联的内存，必须销毁或关闭互斥量。不过，如果互斥量仍然被占有，或者存在阻塞的线程还等待着该互斥量的释放 (release)，就不能销毁或关闭这个互斥量。

四、事件互斥量和条件变量

程序清单 6-2 中的程序使用一个 while 循环来让 `threadA()` 保持繁忙，直到队列中至少存在一个即将处理的项。没有这个繁忙的循环，`threadA()` 可能会尝试从一个空 `TextFiles` 队列中删除某个文件名。主线程需要以某种方式与 `threadA()` 通信，了解队列中存在可以处理的文件名。我们可以使用一个全局标志，允许主线程在队列中存在可以处理的项时设置标志。全局标志存在的问题是，`threadA()` 必须不断地测试标志的值，看队列中是否存在可用的文件名。不断地检查将会占用多处理器上的某处理器。如果不断检查发生在单个处理器机器上，其它线程被中断，将处理器时间给

予繁忙的循环。而且繁忙循环只对列表中的第一个项起作用。如果队列中只包含一个文件名，而且又从队列中删除了，那么，在主线程有机会在队列中放置另一个文件名之前，threadA()将试图删除另一个文件名，此时创建另外的错误条件。显然，对于这一点，我们可以做得更好。

POSIX、Win32 和 OS/2 线程支持一种线程间的广播机制 (broadcast mechanism)，它允许一个线程广播给另一个线程，告诉它某种事件已经发生。这种机制使用的是所谓的条件变量 (condition variable) 或事件互斥量 (event mutex)。当一个线程尝试锁定一个常规互斥量时，它将在这个互斥量上阻塞，直到该互斥量的占有者释放了互斥量，或者经过了指定的时间段为止。当一个线程锁定了一个事件互斥量，它将阻塞，直到它接到了一条广播，告知它可以继续为止。我们可以修改程序清单 6-2 中的程序，让它使用事件信号量。程序清单 6-3 中的程序使用 OS/2 事件信号量来通知 threadA() 队列里面至少有一个文件名。

程序清单 6-3 这个程序使用了两个线程。第一个线程查找包含字符串 txt 的所有文件名。第二个线程搜索这些文件的特定关键字列表。程序使用互斥量来保护临界区，以及使用一个事件互斥量来告知 TextFiles 队列何时至少有一个元素

```

01  #include <dirent.h>
02  #include <set.h>
03  #include <cstring.h>
04  #include <fstream.h>
05  #include <algo.h>
06  #include <string.h>
07  #include <list.h>
08  #include <stack.h>
09  #include <stddef.h>
10  #include <process.h>
11  #define INCL_DOSPROCESS
12  #define INCL_DOSSEMAPHORES
13  #include <os2.h>
14
15  queue< list<string> > TextFiles;
16  set<string,less<string> > Keywords;
17  set<string,less<string> > SearchWords;
18  void threadA(void *X);
19  HMTX SemHandle;
20  HEV SemEventHandle;
21  string InFile;
22  string OutFile;
23
24
25
26  void threadA(void *X)
27  {
28      string Temp;
29      string FileName;

```

```
30     less<string> Comp;
31     ifstream In;
32     ofstream Out;
33     In.open(InFile.c_str());
34     Out.open(OutFile.c_str());
35     while(!In.eof())
36     {
37         In >> Temp;
38         if(!In.eof()){
39             KeyWords.insert(Temp);
40         }
41     }
42     In.close();
43     DosWaitEventSem(SemEventHandle, SEM_INDEFINITE_WAIT);
44     while(!TextFiles.empty())
45     {
46         DosRequestMutexSem(SemHandle, 5000);
47         FileName = TextFiles.front();
48         TextFiles.pop();
49         DosReleaseMutexSem(SemHandle);
50         In.open(FileName.c_str());
51         SearchWords.erase(SearchWords.begin(),
52                             SearchWords.end());
53         while(!In.eof() && In.good())
54         {
55             In >> Temp;
56             SearchWords.insert(Temp);
57         }
58         In.close();
59         if(includes(SearchWords.begin(), SearchWords.end(),
60                     KeyWords.begin(), KeyWords.end(), Comp)){
61             DosRequestMutexSem(SemHandle, 5000);
62             cout << "Thread A match found in " << FileName << endl;
63             Out << FileName << endl;
64             DosReleaseMutexSem(SemHandle);
65         }
66     }
67     Out.close();
68 }
69
70
71 void main(int argc, char *argv[])
72 {
73
74     DIR *DirP;
```

```

75     unsigned long Result = 0;
76     struct dirent *EntryP;
77     DirP = opendir(argv[1]);
78     InFile = argv[2];
79     OutFile = argv[3];
80     if(argc == 4){
81         string Temp;
82         DosCreateMutexSem(NULL,&SemHandle,0,0);
83         DosCreateEventSem(NULL,&SemEventHandle,0,0);
84         Result = _beginthread(threadA,8192,NULL);
85         if(DirP == NULL){
86             cerr << " Could Not Open " << argv[1] << endl;
87             exit(1);
88         }
89         do{
90             EntryP = readdir(DirP);
91             if(EntryP){
92                 Temp = EntryP->d_name;
93                 if(Temp.contains(".TXT")){
94                     DosRequestMutexSem(SemHandle,5000);
95                     cout << "Found " << EntryP->d_name <<
96                        " in Main Thread " << endl;
97                     TextFiles.push(EntryP->d_name);
98                     DosReleaseMutexSem(SemHandle);
99                     DosPostEventSem(SemEventHandle);
100                 }
101             }while(EntryP);
102             closedir(DirP);
103             DosWaitThread(&Result,DCWW_WAIT);
104             DosCloseMutexSem(SemHandle);
105         }
106     }
107 }
108
109
110

```

条件变量 (condition variable) 以及对应的事件互斥量 (event mutex) 都是特殊的变量。在 OS/2 中, 事件互斥量由 `DosCreateEventSem()` 调用创建。在 Win32 中, 事件互斥量通过对 `CreateEvent()` 函数的调用创建。在 POSIX 中, 条件变量通过初始化一个 `pthread_cond_t` 类型的变量来创建, 这个变量可以通过一个缺省值来初始化, 例如:

```
pthread_cond_t EventVariable=PTHREAD_COND_INITIALIZER;
```

或者通过函数 `pthread_cond_init()` 来调用。在 POSIX、Win32 和 OS/2 中, 对事件互斥量和条件变量的可用操作请见表 6-6。

表 6-6 在 POSIX、Win32 和 OS/2 中, 对事件互斥量和条件变量的可用操作

环 境	对事件互斥量的操作	描 述
POSIX	pthread_cond_init()	初始化条件变量
	pthread_cond_destroy()	销毁条件变量
	pthread_cond_wait()	挂起调用线程, 直到另一个线程信号通知条件变量为止
	pthread_cond_timedwait()	挂起调用线程, 直到另一个线程在指定时间内的信号通知条件变量为止。如果没有在时间内的信号通知条件, 则从等待中释放该线程
	pthread_cond_signal()	为等待锁定互斥量的线程创建一个规划次序。将线程锁定的互斥量赋予线程。将所有其它线程放在一个等待获取该互斥量的队列中
	pthread_cond_broadcast()	释放所有等待条件变量的线程。实际上只赋予一个线程互斥量的占有权
OS/2	DosCreateEventSem()	创建事件信号量
	DosOpenEventSem()	打开事件信号量
	DosPostEventSem()	发出事件信号量
	DosResetEventSem()	重置事件信号量
	DosWaitEventSem()	等待事件信号量的发送
	DosCloseEventSem()	关闭事件信号量
	DosQueryEventSem()	返回事件信号量的发送数。发送计数指事件信号发送的次数
Win32	CreateEvent()	创建一个事件信号量对象。如果将某个名字指定为它的参数, 则创建一个命名事件信号量对象
	OpenEvent()	将对象的名字用作它的参数, 从一个无关进程打开一个事件信号量对象
	SetEvent()	将事件信号量对象的状态更改为被通知。如果事件已经被通知, 函数将无作用
	ResetEvent()	重置事件信号量对象。它更改对象的状态为被通知
	PulseEvent()	将对象的状态更改为被通知。释放所有等待线程, 然后将对象的状态更改为未被通知
	WaitForSingleObject()	导致线程一直等待到对象被通知。将对象指定为参数之一
	WaitForMultipleObjects()	导致线程等待一个包含一个或多个同步对象的数组

在每种环境中都必须创建事件互斥量。创建事件互斥量后，就可以对它执行3个主要的操作。事件互斥量可以等待、延后和销毁。下面的代码摘自程序清单 6-3，其中的主线程在给队列添加了一个文件名后，就使用 `DosPostEventSem()` 来广播通知事件的发生：

```
...
...
...
DosRequestMutexSem(SemHandle, 5000);
cout<<"Found" <<EntryP->d_name
    <<"in Main Thread" <<endl;
TextFiles.push(EntryP->d_name);
DosReleaseMutexSem(SemHandle);
DosPostMutexSem(SemEventHandle);
...
...
```

第二个线程 `threadA()` 等待着事件：

```
DosWaitEventSem(SemEventHandle, SEM_INDEFINITE_WAIT);
```

`threadA()` 接收到广播后，它可以在等待后继续。互斥量导致一个线程一直阻塞到该互斥量被释放，事件互斥量（或条件变量）用于让一个或多个线程一直等到条件满足为止，或者等到一个事件或多个事件的发生。常规互斥量只可以用于通信联络锁定和取消锁定。条件变量和事件互斥量可以基于一个或多个事件或条件的发生，协调两个线程间的同步。程序清单 6-3 中程序里的事件就是在 `TextFiles` 队列中插入一个文件名。如果这个事件永远也不发生，那么从理论上讲，`threadA()` 就会无限期地等待。这种情形称做无限延迟（indefinite postponement）（Deital, 1990）。不过，在 OS/2 这样的环境中，可以指定等待条件在一段时间后终止它，这样防止了无限延迟条件。

五、等待多个条件

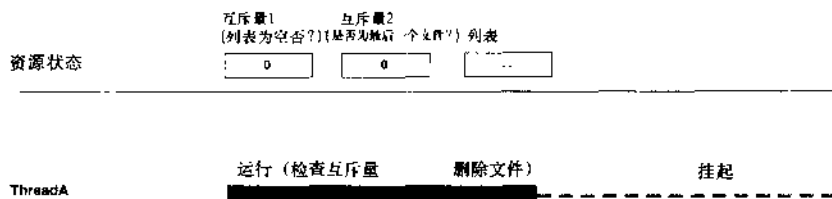
事件互斥量或条件变量的概念可以扩展到包括多个事件或多个条件。也就是说，可以让一个线程同时等待多个变量。程序清单 6-3 中的程序仍然存在缺陷。它只是等待一个文件名被放到队列中。甚至在没有剩余文件名供搜索时，可能 `threadA()` 仍然试图删除文件名。可以将 `threadA()` 设计成等待多个条件。图 6-5 显示了多个事件互斥量或多个条件变量影响 `threadA()` 流程的方式。

6.3.5 临界区

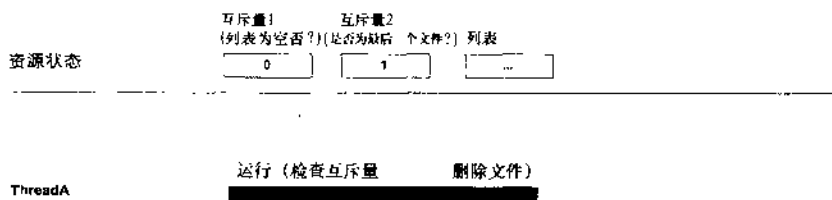
程序的临界区是一个代码块，它可以修改被两个或多个并发执行线程或进程共享的数据或设备状态。例如，正写入同一个文件的两个或多个并发执行线程有一个共同的临界区。临界区位于更改文件指针位置、写入文件，或关闭文件的代码行。如果多个线程试图同时访问一个通信点，通信端口的状态可能被试图将波特率更改为不同值的多个线程破坏。信号量、互斥量、同步变量、条件变量以及事件信号量都可以用于在多线程程序中保护临界区。尽管每个概念在实现上可能不一样，但它们基本上都服务于同一个目的，即在访问临界区时让多个线程或进程同步化。不要将术语“临界区”与 Win32 中使用的临界区数据类型（Critical Section data type）混淆，后者充当一个匿名互斥量，而且作为一个常规互斥量服务于同一个目的。OS/2 环境支持 `DosEnterCritSec()` 函

数，它能有效禁止当前进程中其它所有线程的执行。这个函数表示临界区的绝对保护。不过，它这样做的代价是停止了进程中所有其它线程的执行。甚至那些不影响称做 `DosEnterCritSec()` 进程的线程也被停止了。有时，`DosEnterCritSec()` 提供的这种“赶尽杀绝”是必需的。

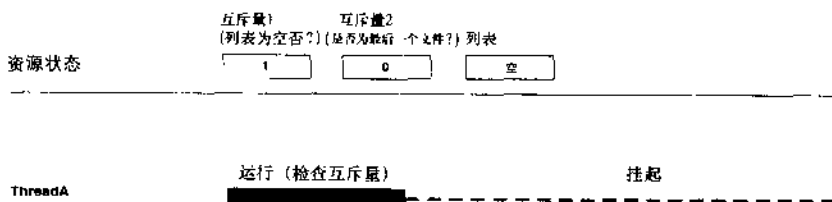
情形 1：列表不为空，而且有更多的文件被添加



情形 2：列表不为空，而且没有更多的文件要添加



情形 3：列表为空，同时有更多的文件要添加



情形 4：列表为空，而且无更多的文件要添加

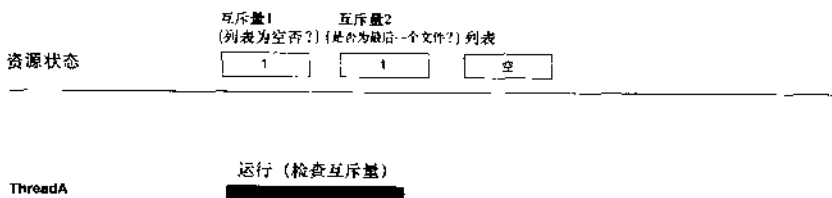


图 6-5 演示多个事件互斥量或多个条件变量如何影响线程 A

无限延迟

对于多线程或并发编程来说，竞争条件 (race condition) 和死锁 (deadlock) 是两个主要的敌人。第三个敌人是无限延迟的发生，它是死锁的一个特例。一般的死锁至少要求两个线程，每个线程占有有一些其它线程希望得到的资源，它只能等到所需资源被释放后才能继续。无限延迟

(indefinite postponement) (有时也称做单进程死锁, one-process deadlock) 并不是必须涉及到资源 (Deitel, 1990)。线程可能等待着某个事件的发生, 或某个条件变成现实, 但情况却是, 事件永远也不会发生, 或者条件永远也不会变成现实。简言之, 无限延迟通常是失败事件的结果, 而不是锁定资源所产生的。例如, 请看下面摘自程序清单 6-3 中的代码片断。

```
DosWaitEventSem(SemEventHandle, SEM_INDEFINITE_WAIT);
while(!TextFiles.empty())
{
    DosRequestMutexSem(SemHandle, 5000);
    FileName = TextFiles.front();
    TextFiles.pop();
    DosReleaseMutexSem(SemHandle);
    In.open(FileName.c_str());
    SearchWords.crase(SearchWords.begin(), SearchWords.end());
    while(!In.eof() && In.good())
    {
        In >> Temp;
        SearchWords.insert(Temp);
    }
    In.close();
    if (includes(SearchWords.begin(), SearchWords.end(),
        Keywords.begin(), Keywords.end(), Comp)) {
        DosRequestMutexSem(SemHandle, 5000);
        cout << "Thread A match found in " << FileName << endl;
        Out << FileName << endl;
        DosReleaseMutexSem(SemHandle);
    }
}
```

以上代码片断受到无限延迟的潜在威胁。下面代码行:

```
DosWaitEventSem(SemEventHandle, SEM_INDEFINITE_WAIT);
```

导致它一直等到特殊事件的发生。一旦事件发生了, 程序将试图锁定这个互斥量并继续下去。不过, 并不能保证程序所等待的事件会发生。这个程序可能被无限延迟, 或者至少需要用 CTRL+C 来终止。当同步变量用于保护临界区时, 它们也会有这样的问题。

6.4 避免竞争条件

Andrew Tanenbaum 在他的 *Operation Systems Design and Implementation* 一书中列出了 4 种用于防止竞争条件的基本技术:

- 不要使用可能同时位于临界区内的两个进程。
- 不要依赖对 CPU 的速度作出的任何假设。
- 不要让临界区外部停止的进程阻塞其它进程。
- 不要让进程等待任意长的时间进入其临界区。

本书的余下内容演示了这些技术, 结合使用了 C++ 中类结构提供的封装来为多线程程序中的

竞争条件提供解决方案。

6.5 死锁必需的条件

死锁在非常特定的条件下发生。事实上，Deitel 已经列出了发生死锁必须存在的 4 个条件 (Deitel, 1990):

- 进程声明排它性控制它们需求的资源。
- 进程在等待其它资源的释放时占有资源。
- 资源不能强行从进程中删除。
- 存在一个循环等待条件。

这些条件的存在是发生死锁必需的，但是不充分的。也就是说，仅仅因为这 4 个条件的存在，我们还不能下结论说一定存在死锁。可能这 4 个条件存在，但不存在死锁。不过，要发生死锁，这 4 个条件必须存在。如果缺少任何一个条件，就不会发生死锁。这为我们防止死锁提供了一个理想的策略。只要至少让一个条件不满足，我们就可以防止死锁的发生。防止死锁是多线程编程中的一个主要目标。死锁的一些先行条件是难以防止的。例如，一个线程或进程必须能够获得对一个内存块或某设备的访问权。这种排它性访问将使用互斥量或信号量来获得。同样，很可能两个进程或线程正占据着其它进程或线程所想要的资源。在多线程编程中，这种情况经常出现。

6.6 远离死锁

也许用于避免死锁的最简单技术是应用定时互斥量、事件变量、同步变量以及信号量。当使用这些机制时，死锁不能发生，因为线程或进程在继续之前只阻塞或等待有限的时间段。例如：

- 1) `DosWaitEventSem(SemEventHandle, SEM_INDEFINITE_WAIT);`
- 2) `DosWaitEventSem(SemEventHandle, 10000);`

使用了第一个调用的线程一直等到事件的发生，即使事件永远也不会发生，它也会等待下去。将等待时间改为一个指定值，就使用了一个定时信号量 (timed semaphore)。调用第二个函数的线程至多等待 10 000 毫秒，然后继续执行一条指令。通过使用定时同步机制，除了发生死锁的 4 个必需条件外，可以压制其它所有条件。还有其它防止死锁的技术，但定时同步机制是最容易理解和实现的。POSIX、OS/2 和 Win32 支持定时同步机制。

就像针对竞争条件一样，我们也可以利用 C++ 对面向编程的支持来管理死锁条件。通过识别导致竞争条件和死锁的情形，防止它们就胜算过半了。通过使用 C++ IPC、ITC 组件以及多线程架构，我们可以去掉应用程序中的竞争条件和死锁。第 8 章将介绍 IPC (进程间控制, interprocess control) 的构造块。

接口类与进程间通信

这意味着，所有到达目的地的途径并无优劣之分。问题在于，某些途径似乎太神秘而让人觉得不可能实现。

The Fourth Dimension

——Rudy Rucker

也许在 C++ 中碰到的最有用类类型 (type of class) 就是接口类 (interface class)。主要原因是，接口类可以用作不同编程方法、数据结构实现技术、编码方式、命名约定以及不同操作系统环境之间的桥梁。事实上，将传统 C 程序移植到对象世界的最简单途径是为 C 程序中自由漂浮的函数和数据提供接口类。当需要具备不同操作系统间或不同硬件平台间的可移植性时，接口类将责无旁贷地成为完成这类工作的武器。

7.1 接口类详解

第 1 章对接口类提供了一个粗略的定义，在这里，我们将对接口类作进一步地详细讨论。接口类用于为代码和数据提供一个新的接口，或更改代码和数据的旧接口。接口类用于修改或改进另一个类或一系列类的接口。在许多情况下，接口类用来给非面向对象编程工作提供面向对象接口，例如为过程库 (procedure library)、操作系统 API，或者数据库管理系统提供面向对象接口。可以使用接口类让另一个类更容易使用，功能更强、更安全，或者在语义上更正确。调整或优化另一个类接口的接口类使原来的类更有用，或者更有效。当为没有面向对象程序提供面向对象特征时，接口类充当现在过程和数据的封装器。接口类封装这些过程和数据，并提供访问它们的成员函数 (member function)。

7.1.1 接口类的类型

接口类分若干类型，每种类型有不同的功能。一些接口类只包含被继承的虚函数 (virtual

function)，而其它接口类为具体类（concrete class），它可以用作类层次的终止点。接口类与其它代码和数据聚集（code and data aggregation）之间的关系如图 7-1 所示。

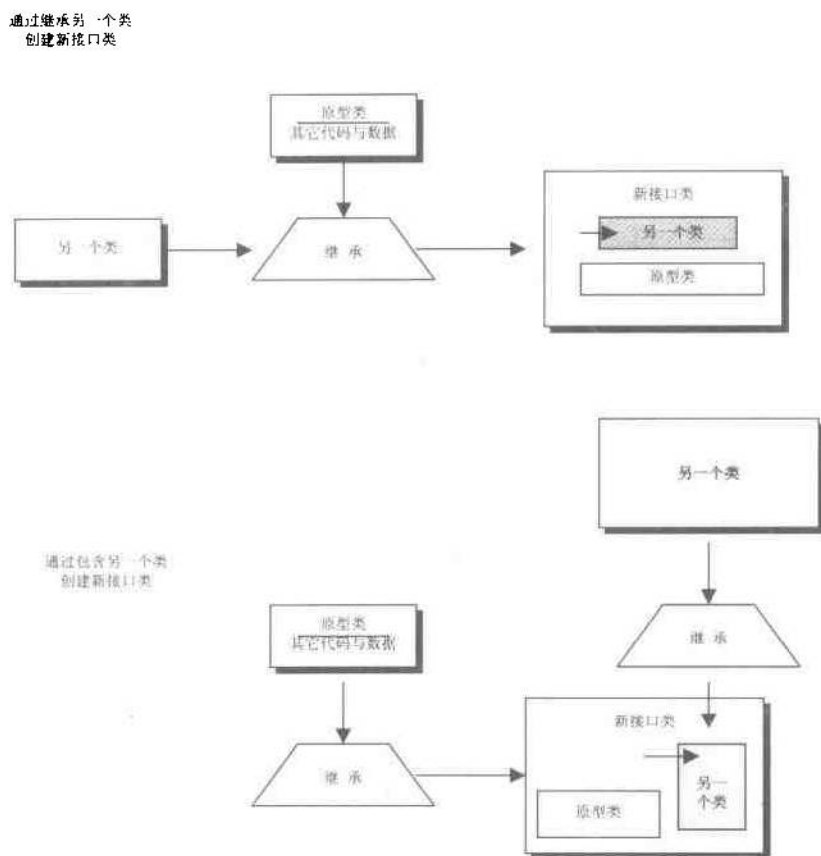


图 7-1 接口类和其它代码与数据聚集的关系。接口类通过继承或封装来实现

一、作为非面向对象数据和代码的封装器

最简单也最有用的接口类是处理非面向对象过程及其数据，并赋予它们面向对象接口的接口类。例如，下面的程序使用一个排序后的数组。程序代码如下所示：

```
#include<stdio.h>
int MyArray[Size];
void getArrayElements()
{
    //执行一些输入
}
void sortArrayElements()
{
    //对 MyArray 执行排序
```

```

    }

    void displayArrayElements()
    {
        //printf(MyArray ...)
    }

    void main(void)
    {
        //对 MyArray 执行操作
    }

```

程序员可以轻易处理这种代码，将它封装成一个 C++ 类。封装成一个类后，以上代码就立即可以利用面向对象技术。为了使用封装器途径，程序员要将程序中的数据封装成一个类，然后只允许这个类的成员函数来访问数据。封装成的类如下所示：

```

class array{
protected:
    int Data[Size];
public:
    array(int);
    void getArrayElements(void);
    void sortArrayElements(void);
    void displayArrayElements(void);
}

```

它封装了小型数组，同时还包括自由飘浮的函数，将它作为一个成员函数。使用这种技术，为非面向对象代码和数据创建一个面向对象接口，或者一个接口类。这样做的优点是显而易见的，最重要的是，类 `array` 中封装的数据现在受保护，只能被属于类 `array` 部分的成员函数来访问。因为已经将数组数据封装成一个类，程序员可以声明类型 `array` 的多个对象。`array` 类现在可以在其它程序中重用（reuse）。通过继承可以特殊化数组，让它包含其它操作等。重要的一点是，通过将过程化代码和数据封装在一个接口类中，让它面向对象化。

当我们搜索构建多线程架构技术时，会使用接口类来封装提供多线程处理服务、进程控制服务、进程间通信、文件 I/O 以及设备 I/O 的操作系统 API。开发出针对操作系统服务的接口类后，就可以不用顾忌特定环境的函数调用、参数以及行为。事实上，接口类的一个重要用途是为数据和需要操作于该数据的函数提供独立域接口（domain-independent interface）。

二、适配器——修改其它类接口的类

另一个接口类类型是适配器类（adaptor class），它为其它现有面向对象类提供封装器。也就是说，适配器类将一个类的接口转换成另一个类接口的样子。适配器类的目标不是添加功能性，而是提供一个新接口。在某些情况下，某个类可能表示我们需要的数据，而且此类包含对数据的基本操作。不过，这个类的接口可能需要调整或重新定义。例如，有一个名叫 `mathematical_expression` 的类，其中有如下所示声明：

```

class mathematical_expression{

```

```

protected:
    String Expression;
    ...
    ...
public:
    mathematical_expression(String X);
    void getMathExpression(String X);
    void tokenizeMathExpression(void);
    ...
    ...
};

```

`mathematical_expression` 类刚好是我们所需要的类。它有一个 `String` 类以及一个将 `String` 类内容解析成符号的操作。不过，我们可能需要在某特定的域使用这个类，但 `getMathExpression()` 和 `tokenizeMathExpression()` 的语义应用在这个域不太准确。例如，如果应用域是自然语言解析，就需要一个具备 `String` 组件的类，而且还能将 `String` 组件解析成符号。通常 `String` 组件并不表示一个数学表达式。在这种场合，就可以使用接口类来调整 `mathematical` 类的接口，让它具有我们需要的接口。我们可以声明一个新类，命名为 `sentence`，并从 `mathematical_expression` 派生出这个类：

```

class sentence:private mathematical_expression{
public:
    sentence(String &S);
    void getSentence(String &S){getMathExpression(S);}
    void parseSentence(void){tokenizeMathExpression(S);}
}

```

现在，对于类 `sentence` 可以使用的操作更适合应用在使用 `sentence` 类的域。请注意，`sentence` 类的作用只是重命名 `mathematical_expression` 类的成员函数。类 `sentence` 不添加任何新数据成员或成员函数（尽管它可以这样做）。它只用于调整 `mathematical_expression` 的接口，让它与自然语言解析域一致。

另一个封装其它类的接口类例子是作为标准模板库（Standard Template Library）一部分的容器适配器。这种适配器为标准模板库中的 `list`、`vector` 和 `deque` 容器提供了一个新的公共接口。例如，程序清单 7-1 中的 `stack` 类用作一个修改 `vector` 类的接口类。

程序清单 7-1 STL stack 类的声明

```

1 template<class Container>
2 class stack{
3 friend bool operator==(const stack<Container>&x,
4                          const stack<Container>&y);
5 friend bool operator<(const stack<Container>&x,
6                        const stack<Container>&y);
7 public:
8     typedef Container::value_type value_type;
9     typedef Container::size_type size_type;
10 protected:

```

```

9   Container c;
10  public:
11   bool empty() const {return c.empty();}
12   size_type size() const {return c.size();}
13   value_type&top() {return c.back();}
14   const value_type&top()const{return c.back();}
15   void push(const value_type& x){c.push_back(x);}
16   void pop(){c.pop_back();}
17  };

```

通过如下声明:

```
stack<vector<T>>>Stack;
```

`stack` 类为用户提供一个语义正确的类。`stack` 的正式概念指的是弹出和推入堆栈 (`stack`) 的操作。堆栈是一种 LIFO (后进先出) 数据结构, 也就是说, 最后推入堆栈的数据片断将最先弹出堆栈。`top` 操作返回下一个被弹出堆栈的项。请注意 13、14、15 和 16 行, `top()`、`push()` 以及 `pop()` 操作只是调用其它成员函数, 如 `c.pop_back()`、`c.push_back()` 以及 `c.back()`。通过调整类 `c` 提供的名字, `stack` 类表示的操作具有用户更加熟悉的名称。在这里, 类 `c` 是一个矢量容器。`stack` 类只用于调整其它类的接口。事实上, `stack` 类可以用于调整其它任何支持函数 `back()`、`push_back()` 以及 `pop_back()` 的容器类型类的接口。

适配的两个途径

使用两种技术将一个类修改成一个新接口。第一种技术使用继承, 如下所示:

```

class sentence:private mathematical_expression{
public:
    sentence(String &S);
    void getSentence(String &S){getMathExpression(S);}
    void parseSentence(void){tokenizeMathExpression(S);}
}

```

`sentence` 类为 `mathematical_expression` 类提供了一个新接口。`sentence` 类使用了私有继承, 这意味着通过指针或对类 `sentence` 的引用不能访问类 `mathematical_expression`。我们不希望用户直接操纵 `getMathExpression()`、`tokenizeMathExpression()` 或任何最终成为 `mathematical_expression` 类元素的组件, 所以, 将它声明为一个私有基类。只允许那些封装了类 `mathematical_expression` 中成员函数的成员函数访问。记住, 一个适配器类可能只选择修改被转换类的部分成员函数。对于所有成员函数都不被修改的情况, 被修改类的私有继承提供了一定程度的安全性。

用于修改类的第二个方法是使用类封存 (containment), 有时称做合成 (composition)。图 7-2 显示了 `sentence` 适配器和 `stack` 适配器的类关系图。`stack` 适配器没有使用继承来获得适配类的实现, 而是使用封存。

```

template<class Container>
class stack{
    friend bool operator==(const stack<Container>& x,
                           const stack<Container>& y;
    friend bool operator<(const stack<Container>& x,
                           const stack<Container>& y;

```

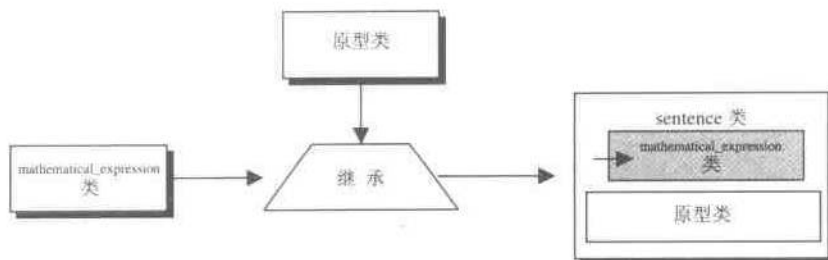


```

public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;

```

sentence 类



stack 类

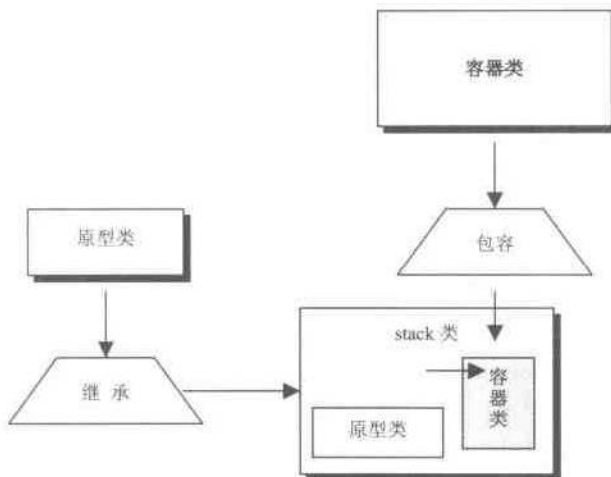


图 7-2 sentence 适配器和 stack 适配器的类关系图

在这里，被封存的是对象 *c*，它是 *Container*（容器）类型。同样，被修改类的成员函数不能通过 *stack* 类的非成员来访问。与 *sentence* 适配器一样，不可能使用指针或引用在多态中访问对象 *c*。只有那些在 *c* 中被赋予了新接口的成员函数才能被访问。在两种情况下，接口类都可以访问修改类，但对于外界，其访问权是有选择性的。使用适配技术，接口类可以使用类修改后的实现来预定所需要的访问。以后通过蓝图接口类（blueprint interface class），我们还会看到，并不是所有的接口类都重命名现在的实现。请注意，若只是重命名适配类的成员函数，这时使用内联（inline）是有利的。使用 C++ 的内联功能，重命名的开销可以降到最低点。

三、蓝图接口类

蓝图接口类 (blueprint interface class) 提供一个接口, 但没有任何实现。这种类用于为将来的类提供基础。典型情况下, 这样的类不包含数据。这种类中的所有的成员函数都为纯虚拟 (pure virtual), 而且这种类的所有基类也为接口类 (Caroll, Ellis, 1995)。这些蓝图接口类计划程序员使用该时必须实现哪些成员函数。这些蓝图接口类不仅要求实现一定的函数, 而且还要求这些函数有一定的命名约定, 因此为接口具有一定的外观提供了保证。如下所示设计一个语言处理器的蓝图类:

```
class parse{
protected:
    String Data;
public:
    virtual bool validateParenthesis(void)=0;
    virtual bool validateTokens(void)=0;
    virtual bool tokenize(void)=0;
    virtual bool evaluate(void)=0;
    virtual bool getData(String &X)=0;
    ...
    ...
    ...
    ...
};
```

虽然语言处理器类没有提供任何实现, 但它为所有派生类提供了最小程度的接口命名约定。具有实例化对象的任何派生类必须定义纯虚函数。作为蓝图提供纯虚函数强加给最终实现的函数, 但纯虚设计对于派生类中的函数实际做什么并没有规定。纯虚函数的意义既显示在与解析类一同提供的文档中, 也显示在命令解析器头文件的注释中。

7.1.2 减小参数和全局变量的数量

使用接口类封装过程代码和数据的一个重要附带作用是减少了调用新成员函数时的参数。例如, 对于 POSIX 线程库中的函数, 大部分都至少要求一个参数。许多还要求多个参数。程序清单 7-2 中程序包含一些常用的 pthread 函数。

程序清单 7-2 本程序使用一个接口类来封装 pthread 组件, 目的是去掉全局变量 MyLock

```
#include <iostream.h>
#include <pthread.h>
#include <rational.h>
#include <set.h>
#include <algo.h>

rational M(3,4);
typedef set<rational,rational> rational_set;
typedef set<rational,rational>::iterator rational_iterator;
```

```
rational_set SetA, SetB, SetC;
rational_iterator A, B, C;

void *threadA(void *X)
{
    rational Z(5,4);
    rational W(1,8);
    SetA.insert(W);
    SetA.insert(Z);
    set_intersection(SetA.begin(), SetA.end(), SetB.begin(),
                     SetB.end(), inserter(SetC, SetC.begin()), Z);
    SetB.insert(W);
    SetA.erase(SetA.begin(), SetA.end());
}

void *threadB(void *X)
{
    rational Q(2,3);
    rational R(5,5);
    SetB.insert(Q);
    SetB.insert(R);
    SetA.insert(Q);
    SetA.insert(M);
    set_union(SetC.begin(), SetC.end(), SetB.begin(), SetB.end(),
              inserter(SetA, SetA.begin()), Q);
}

void main(void)
{
    pthread_t ThreadA, ThreadB;
    pthread_create(&ThreadA, NULL, threadA, NULL);
    pthread_create(&ThreadB, NULL, threadB, NULL);
    pthread_join(ThreadA, NULL);
    pthread_join(ThreadB, NULL);
    A = SetA.begin();
    B = SetB.begin();
    C = SetC.begin();
    cout << "Set A contains ";
    while(A != SetA.end())
    {
        cout << *A << " ";
        A++;
    }
    cout << endl << "Set B contains ";
```

```

while(B != SetB.end())
{
    cout << *B << " ";
    B++;
}
cout << endl << "Set C contains ";
while(C != SetC.end())
{
    cout << *C << " ";
    C++;
}
cout << endl;
}

```

在这个程序中, 通过使用接口类来封装不同的 pthread 组件, 我们可以去掉全局变量 MyLock。可以把对这个变量的访问权限只限制于那些接口类中定义的成员函数。我们可以把与 pthread 特定的调用 pthread_mutex_lock() 和 pthread_mutex_unlock() 修改成更一般性的调用 lock() 和 unlock()。我们可以声明一个类 mutex:

```

class mutex{
public:
    int lock();
    int unlock();
private:
    pthread_t MyLock;
}

```

通过定义这个接口类, 我们在让程序清单 7-2 的程序更具系统独立性的道路上迈进了主要的一步。已经封装了对变量 MyLock 的访问, 让程序更安全, 而且去掉了 lock() 和 unlock() 所需要的参数。当在本章后面内容中讨论封装操作系统 API 时, 我们将更详细地探索接口类的这个用途。

7.2 C++没有多线程处理的关键字

C++是一种设计用于生成 (build) 和重用 (reuse) 的语言。在速度和空间效率上, 它的性能与 C 语言是相等的。所以, 其它语言提供的许多庞大而且开销昂贵的操作环境特征并没有集成到 C++ 中。例如, C++ 语言定义中不存在并行语言结构。不过, 它们都可以在 C++ 生成。事实上, 这是 C++ 语言的强项之一。考虑到 C++ 并没有支持并发或多线程的关键字, 所以这一点很重要。这种不支持是有意的。当添加语言结构来支持并行编程或并发时, 语言结构的设计者最终必须挑选出一些支持并行化的模型。一旦选择了这些模型, 通常就要排除其它模型。如果在 C++ 中添加了关键字“并行(parallel)”, 那这个关键字支持哪一个模型呢? 对并行处理有几种经典的硬件模型:

- 单指令多数据流 (SIMD)。
- 多指令单数据流 (MISD)。
- 多指令多数据流 (MIMD)。
- 单程序多数据流 (SPMD)。

并行处理环境（其中所有的处理器共享单个地址空间）与每个处理器拥有各自地址空间的环境大不一样。请考虑这样一种情形，存在多个处理器、多个地址空间以及多台机器——它们都执行单个程序。试图提供一套语言结构来支持并发的大部分模型，事实证明这种尝试困难重重。在使用 C++ 的程序员中，有如此多要求并行处理的不同类型程序员、而且应用的环境又是如此复杂，因此，仅凭单个模型，甚至用少量的模型，是不可能实现的。于是改而按库的形式实现并发，或多线程编程。按这种途径，可以用库来实现所期望的任何模型，而不必强迫每个 C++ 用户掌握并行编程。

如果 C++ 中置入了并行处理的特定模型，程序员将限制于抽象机器所处的表达力。如果将功能类型建成库，而不是语言结构，那么更改功能的实现就不必修改编译器或虚拟机模型。请记住，经常会引入一些硬件并行化的新模型。如果采用了关键字的途径，C++ 语言的定义就要不断地修改。实质上，将并行处理和多线程处理推到库或类库的高度，C++ 获得的是最灵活的处理方式。当把多线程处理或并发建立为库，程序员可以“即插即用”不同的并发模型。

因为 C++ 没有支持多线程处理或进程间通信的关键字，所以，我们必须用类、类库和应用框架的形式构建这种支持。如果我们在 C++ 中构建多线程架构，我们必须从零开始构建线程和任务库、重用现有库，或两者兼有之。因为 POSIX 线程标准的引入，导致在大部分主要 UNIX 环境中线程包的实现，而且因为 OS/2 和 Win32 环境也支持线程，所以，我们可以重用操作系统代码来协助构建 C++ 线程处理和任务处理组件。本书所使用的唯一并发模型就是由操作环境的多线程处理功能所提供的。我们选择重用操作系统代码。在 C++ 类或类库中使用操作系统代码是一种典型，而且切合实际的代码重用形式。我们可以设计 IPC 类、线程控制类以及进程控制类，让它们都重用操作系统代码。这些类最终生成一个接口类。使用这些接口类，我们可以构建任务和线程类。使用任务和线程类，我们可以构建线程和任务库。使用线程和任务库，我们可以构建多线程应用框架。

7.3 面向对象接口到管道

两种可用的最基本 IPC 机制是匿名管道（anonymous pipe）和命名管道（named pipe）。Win32、OS/2 和 UNIX 支持这两种管道。表 7-1 列出了在 Win32、OS/2 和 UNIX 环境中支持匿名管道和命名管道的 API。

表 7-1 Win32、OS/2 和 UNIX 环境中支持匿名管道和命名管道的 API

环 境	命名管道函数调用	描 述
UNIX	mknod()	用于创建一个命名管道
	chmod()	用于更改管道的权限
	close()	用于关闭管道的读或写终端
	open()	用于打开管道的读或写终端

续表

环 境	命名管道函数调用	描 述
UNIX	write()	用于在管道中写入数据
	read()	用于从管道读取数据
	mkfifo()	创建一个命名管道, 甚至在所有进程关闭它后仍然存在
OS/2	DosCallNPIPE()	写入双重消息管道、从中读取, 然后关闭该管道
	DosCreateNPIPE()	创建一个命名管道
	DosDisconnectNPIPE()	客户进程已经关闭命名管道的通知
	DosConnectNPIPE()	通过将管道设置成侦听状态, 为客户进程准备一个命名管道
	DosPeekNPIPE()	检查命名管道中的数据, 而不从管道中取出数据。它返回管道状态的信息
	DosQueryNPIPEInfo()	返回命名管道的信息
	DosQueryNPIPESemState()	返回附加于一个 muxwait 信号量和一个共享事件上的本地命名管道的信息
	DosSetNPHState()	重置命名管道的实际阻塞模式
	DosSetNPIPESem()	在本地命名管道上附加一个共享事件信号量
	DosTransactNPIPE()	写入双重消息管道, 然后从命名管道中读取
	DosWaitPipe()	等待命名管道的某个实例变成可用
Win32	CreateNamedPipe()	服务器进程使用它在本地机器上创建一个命名管道, 并返回一个用于访问该管道的句柄
	CreateFile()	客户进程使用它在本地机器上创建一个命名管道
	CallNamedPipe()	客户进程使用它连接到一个命名管道实例, 然后写入一条消息、读取消息, 并关闭管道句柄。它只能被消息管道使用
	PeekNamedPipe()	读取通过管道传输的数据, 而不会从字节管道或消息管道取出数据。它返回管道实例的信息
	ConnectNamedPipe()	客户进程使用它连接到一个现有命名管道
	DisconnectNamedPipe()	客户和进程使用完命名管道实例后, 服务器进程调用此函数。它关闭到客户进程的连接。客户句柄变成无效, 并抛弃管道中的所有未读数据
	TransactNamedPipe()	写请求消息, 并读答复消息。如果将调用进程的管道句柄设置成消息读取模式, 就可以与消息管道一起使用

续表

环 境	命名管道函数调用	描 述
	FlushFileBuffers()	服务器进程调用该函数确保客户进程读取写入命名管道的所有字节和消息。如果函数没有返回到客户进程，它就从管道读取了所有的数据
	GetNamedPipeInfo()	获取命名管道的信息。它返回管道的类型、输入和输出缓冲器的大小，以及可以创建的管道实例的最大数
	GetNamedPipeHandleState()	提供句柄，包括管道的读和等待模式、管道实例的当前数，以及通过网络进行通信的管道的其它所有重要信息
	SetNamedPipeHandleState()	设置管道句柄的读和等待模式。它还控制搜集字节的最大数，或者客户进程与远程服务器通信时传输消息前等待的最长时间
	WriteFile()	写入一个命名管道。它与字节和消息管道一起使用。它使用事件对象来通知其完成
	ReadFile()	从命名管道读。它与字节和消息管道一起使用。它使用事件对象来通知其完成

匿名管道的基本功能在每种环境中都是相同的，但命名管道在 Win32 和 OS/2 中的功能与大部分 UNIX 环境中的功能大相径庭，记住这一点很重要。虽然这些环境都支持管道，但系统 API 对于这些机制来说不是面向对象的，因此不支持封装和继承。如果我们要使用操作系统 API 来构建多线程应用框架的基石，必须首先将必需的操作系统服务引入对象世界。第 5 章中对管道的讨论表明，管道是一个访问方式像序列文件一样的数据结构。它形成了两个进程间的通信渠道。管道结构通常使用文件读和写方式来访问。如果进程 A 希望通过管道给进程 B 发送数据，则进程 A 将数据写入管道。进程 B 为了接收数据，它必须读取管道。

在基于操作系统服务构建面向对象管道结构之前，我们必须能够把管道的概念描述成一种抽象数据类型。我们要能声明它包含数据组件、对这些数据组件能执行哪一系列操作或服务。在决定了管道应当包含的基本内容后，再考虑系统 API 可以支持什么样的数据结构和服。

为了获得抽象管道的概念，我们从管道共用的基本特征和行为开始。将管道描述为两个或更多进程间的一种通信渠道。为了让进程能通信，必须在它们之间传输某些类型的信息。这些信息可能代表将要执行的数据或命令。不管数据表示什么内容，当它们从一个进程传输到另一个进程时，必须存在保存它们的地方。我们将保存信息的存储区域称做数据缓冲器 (data buffer)。面向对象管道处理方式必须同时支持将数据保存到数据缓冲器的方法，以及从数据缓冲器中读取数据的方法。这些操作称做插入 (insertion) 和提取 (extraction)。在进行数据缓冲器的数据插入或提取之前，这个数据缓冲器必须首先存在。因此，面向对象管道处理方式必须支持创建数据缓冲器的操作。一旦进程间的通信完成，就不再需要保存信息的数据缓冲器。任何面向对象处理方式至

少具有 5 个组件：

- 数据缓冲器；
- 数据缓冲器插入操作；
- 数据缓冲器提取操作；
- 数据缓冲器创建操作；
- 数据缓冲器析构操作。

显然，我们创建的所有管道都具备这 5 个组件。但仅有这 5 个组件并不能表示一个管道。管道有两个终端。一个管道终端用于插入数据，另一个终端用于提取数据。这两个终端可从不同的进程使用。通过这两个终端，简单的管道抽象可以被看作任何类型的 I/O 组件。这种抽象可以用于描述一个列表、队列、矢量、双端队列等等。所以抽象工作还没有完成。为了抓住管道概念的实质，我们必须包含输入和输出端口，这些端口能与单独的进程相连。这样，描述面向对象管道就有 7 个基本组件：

- 输入端口；
- 输出端口；
- 数据缓冲器；
- 数据缓冲器插入操作；
- 数据缓冲器提取操作；
- 数据缓冲器创建操作；
- 数据缓冲器析构操作。

这些组件是描述抽象管道的最小需求。一旦具备以上基本组件，我们就可以确定现在操作系统 API 或数据结构如何用于帮助我们设计面向对象管道。我们将探讨两种创建面向对象管道的技术。第一种技术使用面向对象工具，`iostream` 层次已经为我们提供了这些工具。第二种技术使用复合和 `fstream` 对象来创建 `pstream` 对象（管道类）。

使用 `iostream` 进行面向对象管道处理

如果我们查看表示管道概念所需要的 7 个基本组件就会发现，其中有 5 个组件是许多基本 I/O 数据结构和容器类型所共有的。事实上，大部分系统文件服务都支持：

- 数据缓冲器；
- 数据缓冲器插入操作；
- 数据缓冲器提取操作；
- 数据缓冲器创建操作；
- 数据缓冲器析构操作。

在 C++ 类中，我们可以通过封装这些服务，而且构建面向对象版本的 I/O 服务来利用这种支持。这样，我们就可以向前迈进一大步。C++ 标准库包含 `iostream`，它是提供输入和输出流面向对象模型的类库。而且，这个面向对象库支持数据缓冲器概念以及所有数据缓冲器相关操作。图 7-3 是 `iostream` 类层次的一个简单类关系图。

`iostream` 类层次的主要组件可以描述为 3 种类：缓冲器组件（buffer component）、翻译组件

(translation component) 以及状态组件 (state component) (Hughes, Hamilton, Hughes, 1995)。缓冲器组件用作传送中字节的保存区。翻译组件负责将匿名字节序列翻译成适当数据类型和数据结构, 同时将数据结构和数据类型翻译成匿名字节序列。翻译组件负责为程序员提供字节流信号量, 其中的所有 I/O, 不论其源 (source) 与目标 (destination) 是哪里, 都被看作字节流 (stream of bytes)。状态组件封装面向对象流的状态。状态组件显示可用于缓冲器组件中数据字节的格式类型。状态组件还显示流是否在追加 (append)、创建、排它性读 (exclusive read) 或排它性写 (exclusive write) 模式中已经被打开; 或者数字是否被解释成十六进制、八进制或二进制。状态组件还可用于判断对于缓冲器组件的 I/O 操作错误状态。通过查询状态组件, 程序员可以判断缓冲器组件的状态是否良好。以上 3 个组件是对象 (object), 可以一起使用形成一个完全面向对象流 (complete object-oriented stream), 或单独作为其它任务中的支持对象。

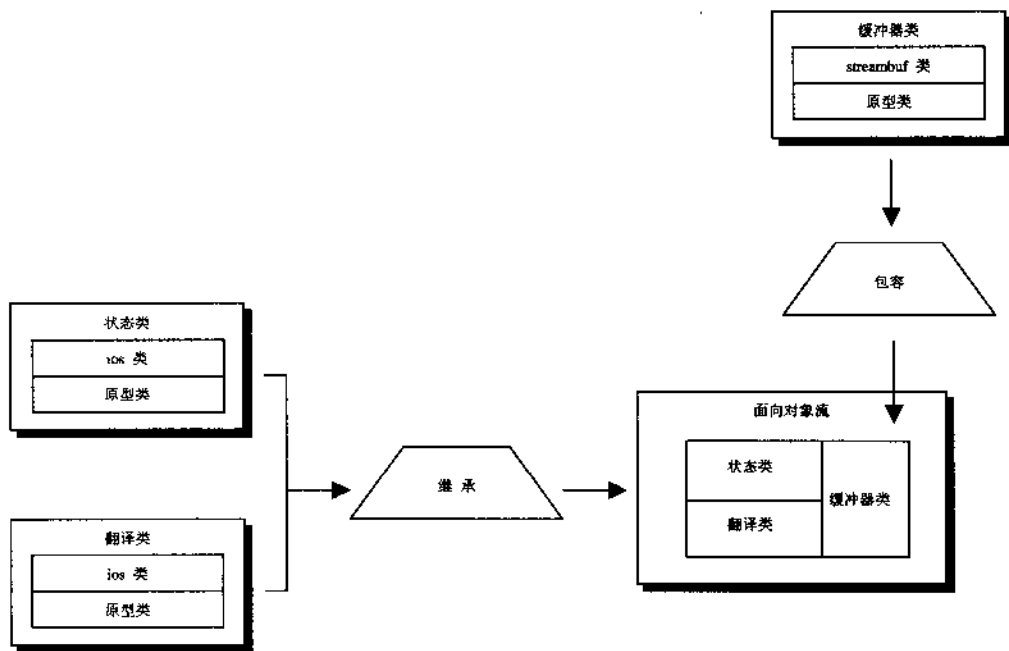


图 7-3 iostream 类层次的高层类关系图

将 `iostream` 当成一个选项后, 实现面向管道的目标就完成了一大半。抽象管道的 5 个基本组件在 `iostream` 中已经实现。我们只需要决定如何才能结合逻辑输入和输出端口的概念与 `iostream`。为了实现这一点, 我们可以考察支持管道应用的系统服务。下面是对缺失端口组件的一个解决方案。例如, 创建一个管道的 UNIX 系统调用:

```
int fd[2];
pipe(fd);
```

`pipe` 函数调用用于创建一个可在父和子进程间通信中使用的管道数据结构。如果对管道的调用成功, 它将返回两个文件描述符。打开 `fd[0]` 用于读, 打开 `fd[1]` 用于写。一旦创建了这两个文

件描述符，它们就可以用于常规的 `read()` 和 `write()` 函数。`write()` 函数将导致数据通过 `Fd[1]` 被插入管道，`read()` 函数导致数据通过 `Fd[0]` 从管道中被提取。因为 `pipe()` 函数返回文件描述符，所以，可以使用系统文件服务来完成管道访问。这两个文件描述符分别表示逻辑输入端口和输出端口。这两个文件描述符提供了对 `iostream` 类库的另一种链接。

`iostream` 类的缓冲器组件有 3 个类家族。表 7-2 列出了 3 种缓冲器类类型及其描述。

表 7-2 3 种缓冲器类类型及其描述

缓冲器类	描 述
<code>streambuf</code>	内存区域集，它具有大量定义内存区域行为的受保护方法。 <code>streambuf</code> 提供了与输入、输出设备间发送数据的接口
<code>strstreambuf</code>	继承了 <code>streambuf</code> 类。它定义了内存缓冲的基本行为。缓冲器按 FIFO 列表或字节数组实现
<code>filebuf</code>	继承了 <code>streambuf</code> 类。它为同文件间的输入或输出提供了一个缓冲器。 <code>get/put</code> 指针是一个指示当前读取或写入信息位置的指针

我们感兴趣的是 `filebuf` 类。`streambuf` 类在标准输入和标准输出 I/O 中用作一个面向对象缓冲器，`strstreambuf` 类用作面向对象内存缓冲器，`filebuf` 类用作文件面向对象缓冲器。检查 `filebuf` 类的接口，以及它的翻译器类 `ifstream`、`ofstream` 和 `fstream` 的接口，我们可以找到一种连接到文件描述符的途径，这些文件描述符来自对 `iostream` 对象的 `pipe()` 系统调用。图 7-4 是 `fstream` 类家族的类关系图。

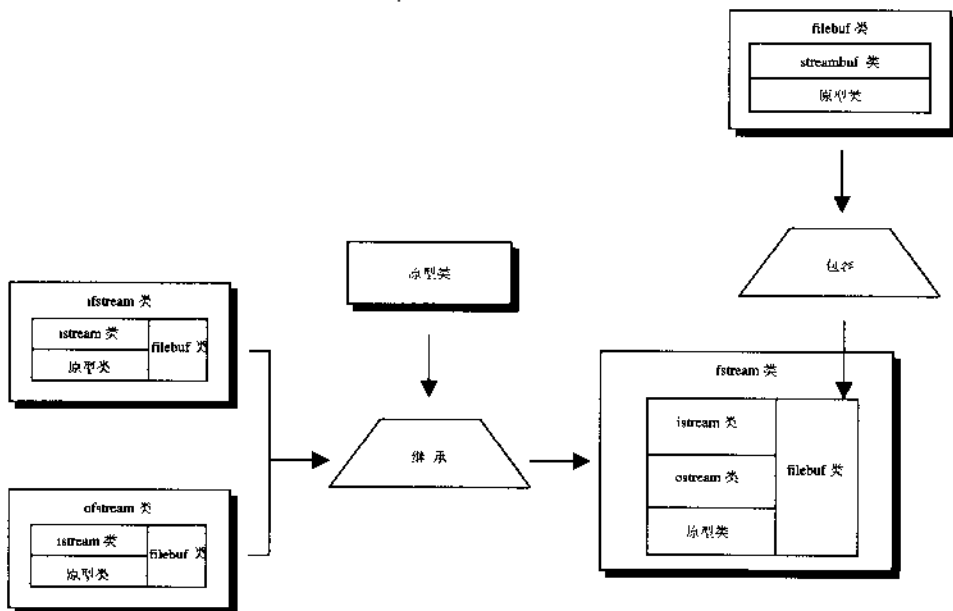


图 7-4 `fstream` 类家族的详细类关系

请注意, `ifstream` 和 `ofstream` 两者都包含 `filebuf` 类。`fstream` 类继承了 `ifstream` 和 `ofstream`, 所以它也包含 `filebuf` 类。因此, 我们可以使用 `fstream` 类家族的任何一个类来帮助我们创建面向对象管道工具。我们可以通过构造函数, 或通过 `attach()` 成员函数连接 `pipe()` 系统调用返回的文件描述符。

一、通过构造函数与 `iostream` 对象连接

有 3 个 `iostream` 类可用于与管道连接: `ifstream`、`ofstream` 和 `fstream`。`ifstream` 对象用于输入, `ofstream` 对象用于输出, `fstream` 对象之中包含 `ifstream` 和 `ofstream` 对象。所以, `fstream` 对象既可以用于输入, 也可以用于输出。`ifstream`、`ofstream` 和 `fstream` 对象有两个构造函数, 可用于接受 `pipe()` 系统调用产生的文件描述符。表 7-3 中的每个构造函数都可以用于构建一个与管道连接的 `iostream` 对象。

表 7-3 `ifstream`、`ofstream` 和 `fstream` 对象构造函数

类	构造函数
<code>fstream</code>	<code>fstream(int fd, signed char *buf, int buf_len)</code> <code>fstream(int fd, unsigned char *buf, int buf_len)</code>
<code>ofstream</code>	<code>ofstream(int fd)</code> <code>ofstream(int fd, char *buf, int len)</code>
<code>ifstream</code>	<code>ifstream(int fd)</code> <code>ifstream(int fd, char *buf, int buf_len)</code>

二、使用 `attach()` 连接 `iostream` 对象与管道

用于连接 `ifstream`、`ofstream` 或 `fstream` 对象与管道的第二种方法是 `attach()` 成员函数, 它被文件描述符作为一个参数来调用。例如:

```
void main(void)
{
    int Fd[2];
    ofstream out;
    pipe(Fd);
    Out.attach(Fd[1]);
    ...
    ...
    //进程间通信
    Out.close();
}
```

`Out.attach(Fd[1])` 调用给管道文件描述符附加一个 `ofstream` 对象。现在, 插入 `Out` 对象的任何信息实际上都写入管道。使用适当的构造函数或 `attach()` 成员函数, 程序员可以连接到管道, 于是通过 `iostream` 类库完成了面向对象进程间通信。

三、使用 `iostream` 的匿名管道

没有关联文件名的管道称做匿名管道 (anonymous pipe)，它只能用于父进程和子进程间。由于子进程自动继承父进程的文件描述符，所以子进程不必引用任何特定文件名就可以使用管道。注意在 `pipe()` 调用中，只有一个对文件描述符的引用：

```
int Fd[2];
pipe(Fd);
```

这个调用创建了一个匿名管道。这里演示的 `pipe()` 调用适用于 UNIX 环境。Win32 和 OS/2 环境也支持匿名管道。表 7-4 列出了在 UNIX、Win32 和 OS/2 环境中创建匿名管道需要的 API。

表 7-4 在 UNIX、Win32 和 OS/2 环境中创建匿名管道需要的 API

环 境	匿名管道函数调用	描 述
UNIX	<code>pipe()</code>	创建一个匿名管道并返回两个文件描述符。保存在 <code>fd[0]</code> 中的描述符与管道的读终端关联。保存在 <code>fd[1]</code> 中的描述符与管道的写终端关联
	<code>close()</code>	用于关闭管道的读或写终端关联
	<code>open()</code>	用于打开管道的读或写终端关联
	<code>write()</code>	用于写入管道数据
	<code>read()</code>	用于从管道读取数据
OS/2	<code>DosCreatePipe()</code>	创建一个匿名管道
	<code>DosOpen()</code>	打开一个匿名管道用于读或写
	<code>DosClose()</code>	关闭匿名管道
	<code>DosDupHandle()</code>	返回一个打开匿名管道的新句柄
	<code>DosRead()</code>	从匿名管道读
	<code>DosWrite()</code>	写入匿名管道
Win32	<code>CreatePipe()</code>	创建一个匿名管道
	<code>WriteFile()</code>	写入匿名管道
	<code>ReadFile()</code>	从匿名管道读
	<code>GetStdHandle()</code>	用于决定进程的当前标准输入和输出句柄
	<code>SetStdHandle()</code>	用于重定向进程的标准输入和输出句柄
	<code>DuplicateHandle()</code>	创建管道句柄的一个非继承拷贝
	<code>CloseHandle()</code>	关闭匿名管道的句柄

创建匿名管道后，该管道可以连接到 `iostream`，然后 `iostream` 可以用于父进程和子进程间的

通信。

使用提取器和插入器进行自动格式翻译是使用 `fstream` 类家族与管道通信的一个主要优点。使用用户自定义提取器和插入器的能力克服了管道编程中碰到的一些困难。`ios` 组件所保留的信息在决定管道状态时有用。当把数据插入管道的一个终端和从另一个终端提取数据时,可用 `iostream` 的翻译组件来执行自动转换。使用管道与 `iostream` 还允许程序员在管道进程间通信中集成 `blob`¹、STL 容器以及算法。图 7-5 显示了使用 `iostream` 进行进程间通信时, `ifstream`、`ofstream`、提取器、插入器以及管道之间的关系。

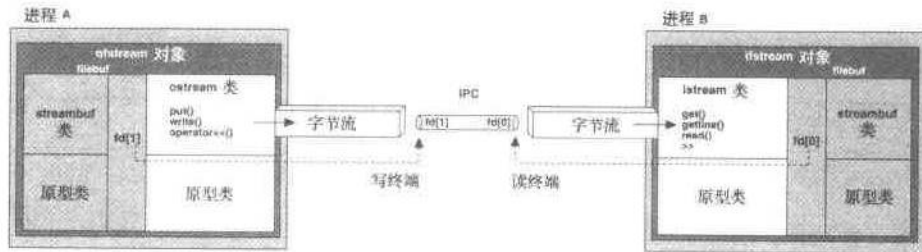


图 7-5 `ifstream`、`ofstream`、提取器、插入器以及管道之间的关系

`fstream` 类家族也可以使用 `read()` 和 `write()` 成员函数来读取管道数据和写入管道数据。程序清单 7-3 中的程序使用 `iostream` 类来执行父进程与子进程之间的进程间通信。

程序清单 7-3 创建一个管道, 然后创建一个子进程。将管道附加到 `fstream` 对象上进行进程间的通信

```

1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <fstream.h>
5  #include <rational.h>
6
7
8  void main(void)
9  {
10     int Fd[2];
11     pipe(Fd);
12     ifstream In;
13     ofstream Out;
14     rational X(1,2);
15     rational Y(3,7);
16     rational Z;
17     double U = 3.4;
18     double Q;
```

¹ 译者注: `blob` 指大型二进制数据类型, 如声音和图像。

```

19     int Pid;
20     Pid = fork();
21     if(Pid == 0){
22         close(Fd[1]);
23         In.attach(Fd[0]);
24         In >> Z >> Q;
25         Z = Z + Y;
26         In.close();
27         cout << Z << " " << Q << endl;
28     }
29     else{
30         close(Fd[0]);
31         Out.attach(Fd[1]);
32         Out << X << " " << U << endl;
33         Out.close();
34     }
35 }
36
37 }
38
39

```

让我们分解以上程序，看看 `iostream` 是如何用于 IPC 的。使用 `pipe()` 系统调用创建管道。创建管道后，再创建子进程。因为子进程是在 `pipe()` 创建后创建的，所以子进程继承父进程的文件描述符。描述符包含管道句柄。子进程使用 UNIX `fork()` 命令创建。如果调用 `fork()` 返回的值为 0，就表示我们处于子进程中。创建两个进程后，开始建立 `iostream` 连接。如下代码：

```

in.attach(Fd[0]);
Out.attach(Fd[1]);

```

用于分别将管道附加到 `ifstream` 和 `ofstream` 对象上。`iostream` 对象连接到管道上后，输入和输出可以继续照常进行。明显的区别在于，`iostream` 不是读取标准输入或写入标准输出，而是在两个进程间通信。下面摘自程序清单 7-3 中的代码块演示了如何使用 `iostream` 来把用户自定义对象插入到管道中：

```

if(Pid == 0){
    close(Fd[1]);
    In.attach(Fd[0]);
    In >> Z >> Q;
    Z = Z + Y;
    In.close();
    cout << Z << " " << Q << endl;
}

```

在以上代码块中，`Z` 是一种用户自定义类型，名叫 `rational`，`Q` 是一个 `double` 类型数值。下面代码行：

```
In>>Z>>Q;
```

从管道中提取 `Z` 和 `Q`。这是使用面向对象管道相对于传统 `read()` 和 `write()` 函数的另一个优点。仅

用这一行就从管道中提取了两种不同类型的对象，因此程序员省去了编写数据类型格式指令的麻烦。任何定义了>>提取运算符的类都可以从管道中提取，而无需程序员做其它进一步的工作。同样，下面代码行：

```
Out<<X<<" "<<U<<endl;
```

用于在管道中插入对象。两样，程序员也不必指定任何数据类型数据大小的信息。<<插入运算符负责进行翻译。注意程序清单 7-3 中的 rational 对象从管道中提取出来后，可用于常规操作。这意味着使用 iostream 进行进程间通信可以让进程通过管道发送对象。在这个例子中，表示分数 1/2 的 rational 对象通过管道被发送。

四、使用 STL ostream_iterator 和 istream_iterator 的匿名管道

我们也可以通过 ostream_iterator 和 istream_iterator 来使用管道。这些迭代器是一般性、面向对象的指针。表 7-5 列出了适用于 ostream_iterator 和 istream_iterator 的一套操作。

表 7-5 适用于 ostream_iterator 和 istream_iterator 的一套操作

迭 代 器	操 作	描 述
istream_iterator	a==b	等价关系
	a!=b	非等价关系
	*a	反引用
	++r	前增量
	r++	后增量
ostream_iterator	++r	前增量
	r++	后增量

这些迭代器通常与 iostream 和 STL 算法一起使用。istream_iterator 对象通常用作序列只读迭代器。也就是说，程序员不应通过 istream_iterator 修改对象。同样，ostream_iterator 是一个序列只写迭代器。两种迭代器都是序列化的。一旦访问了某个项，如果不重新开始迭代，程序员就不能返回来访问它。这意味着，seek() 类型操作不能与 ostream_iterator 和 istream_iterator 一起使用。这些迭代器能与管道结合使用的原因是迭代器与 iostream 之间的连接。图 7-6 显示了 STL I/O 迭代器与 iostream 类之间的关系。它还显示了这些类如何与管道交互的。

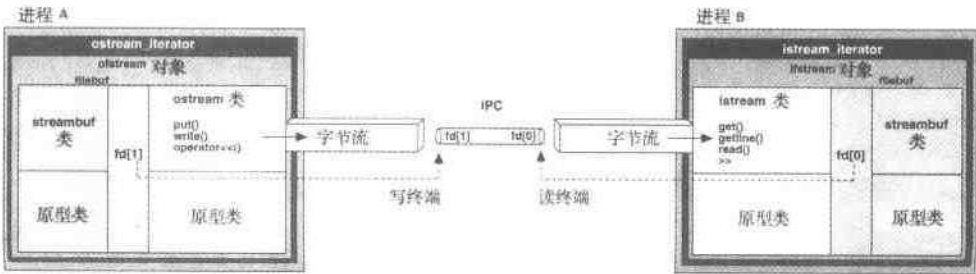


图 7-6 STL I/O 迭代器与 iostream 类之间的关系，以及这些类如何与管道交互的

让我们进一步看看 `ostream_iterator` 是如何用于一个 `ostream` 对象的。如果指针被推进，我们希望它指向内存中的下一个位置。当 `ostream_iterator` 被推进时，它移动或指向输出流中的下一个位置。当给反引用指针赋值时，就是将该值放到指针指向的位置。当我们给 `ostream_iterator` 赋值时，就是将该值放到输出流中。如果这个输出流与 `cout` 连接，则这个值将显示在标准输出上。如果我们按下面方式声明一个 `ostream_iterator` 对象：

```
ostream_iterator<int>X(cout, " ");
```

那么，`X` 就是一个 `ostream_iterator` 类型的对象。增量操作：

```
X++;
```

导致 `X` 移到输出流中的下一个位置。下面语句：

```
*X=Y;
```

导致 `Y` 在标准输出显示，因为赋值运算符`=`被重载用来使用一个 `ostream` 对象。这个运算符的实现出现在 STL 头 `iterator.h` 中，工作方式如下：

```
ostream_iterator<T>&operator=(const T& value)
{
    *stream<<value;
    if(string) *stream<<string;
    return *this;
}
```

它看起来像发生了赋值，但实际上是将值插入到一个流对象中。在这里就指 `cout`。如下声明：

```
stream_iterator<int>X(cout, " ");
```

导致 `X` 通过 `cout` 构建为流。构造函数中的另一个参数是自动放在插入流中的每个 `int` 后面的分隔符。`ostream_iterator` 声明如下所示：

```
template<class T>
class ostream_iterator: public output_iterator{
protected:
    ostream* stream;
    char* string;
public:
    ostream_iterator(ostream& s):stream(&s), string(0){}
    ostream_iterator(ostream& s, char* c):stream(&s), string(c){}
    ostream_iterator<T>& operator=(const T& value){
        *stream<<value;
        if(string) *stream<<string;
        return *this;
    }
    ostream_iterator<T>& operator*(){return *this;}
    ostream_iterator<T>& operator++(){return *this;}
    ostream_iterator<T>& operator++(int){return *this;}
};
```

`ostream_iterator` 的构造函数接受一个对 `ostream` 对象的引用。`ostream_iterator` 类与 `ostream` 类存在一种包含的关系。图 7-7 显示 `ostream_iterator` 类的关系。

`istream_iterator` 与 `ostream_iterator` 的用途相反。它通过 `istream` 对象来使用，而不是通过 `ostream`

对象。ostream_iterator 对象的赋值运算符最终导致如下语句被执行：

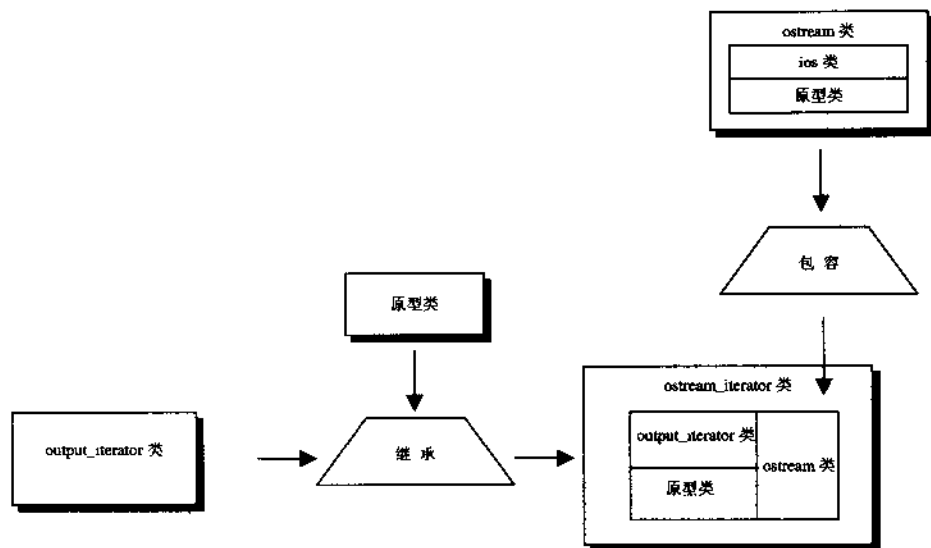


图 7-7 ostream_iterator 类的关系图

```
*stream<<value;
```

当使用 istream_iterator 时，赋值运算符最终导致如下语句被执行：

```
*stream>>value;
```

这意味着，istream_iterator 和 ostream_iterator 都依赖于 ostream 和 istream 类的提取器和插入器操作。所以，对 istream 和 ostream I/O 函数，调用对应 istream_iterator 和 ostream_iterator 对象的机制。如果 istream_iterator 和 ostream_iterator 对象都与 iostream 对象连接，iostream 对象反过来又与管道文件描述符连接，每次推进 istream_iterator 时就读取管道，每次推进 ostream_iterator 时就写入管道。

程序清单 7-4 中的程序使用了 iostream 类、STL 转化算法以及 ostream_iterator 来演示如何在进程间通信中使用迭代器类。

程序清单 7-4 创建 4 个双精度数并插入到一个 set 中。创建一个子进程。父进程创建一个 ostream 迭代器，并使用 STL 转换算法将 set 中的每个元素一分为二，将结果写入管道

```

1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <fstream.h>
5  #include <rational.h>
6  #include <iterator.h>
7  #include <set.h>
8  #include <algo.h>
  
```

```
9
10 set<double,less<double> > SetA;
11 set<double,less<double> >::iterator SItr;
12
13 double Half(double &X)
14 {
15     double Y = 0.5;
16     double Z;
17     Z = X * Y;
18     return(Z);
19 }
20 void main(void)
21 {
22     int Fd[2];
23     ifstream In;
24     ofstream Out;
25     double X = 14.2;
26     double Y = 6.2;
27     double M = 24.12;
28     double Z = 1000;
29     SetA.insert(X);
30     SetA.insert(Y);
31     SetA.insert(M);
32     SetA.insert(Z);
33     int Pid;
34     pipe(Fd);
35     Pid = fork();
36     if(Pid == 0){
37         close(Fd[1]);
38         In.attach(Fd[0]);
39
40         SItr = SetA.begin();
41         while(SItr != SetA.end())
42         {
43             In >> Z;
44             cout << *SItr << " halved is " << Z << endl;
45             SItr++;
46         }
47         In.close();
48     }
49     else{
50         close(Fd[0]);
51         Out.attach(Fd[1]);
52         ostream_iterator<double> Itr(Out," ");
53         transform(SetA.begin(),SetA.end(),Itr,Half);
54         Out.close();
```

```

55
56     }
57
58 }
59
60

```

程序清单 7-4 中的程序包含 4 个双精度数：X、Y、M 以及 Z。给它们赋值，然后将它们插入 SetA。接着程序通过 UNIX fork() 系统调用创建了一个子进程。父进程创建一个名叫 Itr 的 ostream_iterator。然后，父进程使用 STL 转换算法用 2 除 SetA 中的每个元素。转换算法操作元素后，使用 Itr 将它写入管道。转换算法的实现如下所示：

```

template<class InputIterator, class OutputIterator,
        class UnaryOperation> OutputIterator
transform(InputIterator first,
          InputIterator last,
          OutputIterator result,
          UnaryOperation op){
while(first!=last) *result++=op(*first++);
    return result;
}

```

执行下面语句：

```
*result++=op(*first++)
```

导致从 op() 返回的值赋予给 result。因为 result 是一个 ostream_iterator，它与一个 ofstream 对象相连，ofstream 对象反过来又与一个管道文件描述符相连，所以，赋值语句导致某个值被写入管道。下面是程序清单 7-4 的输出结果：

```

6.2 halved is 3.1
14.2 halved is 7.1
24.12 halved is 12.06
1000 halved is 500

```

这个程序使用了 ostream_iterator、transform() 算法和 iostream 类在父进程和子进程之间执行进程间通信。一般而言，接受 input_iterator 或 output_iterator 的任何 STL 算法都可以应用于进程间通信事务。表 7-6 列出了可以使用输入或输出迭代器的 STL 算法。

表 7-6 可以使用输入或输出迭代器的 STL 算法

迭 代 器	操 作
istream_iterator	accumulate
	count
	count_if
	find
	find_if
	for_each

续表

迭 代 器	操 作
istream_iterator	equal includes inner_product lexicographical_compare max_element min_element mismatch partial_sort_copy
ostream_iterator	rotate_copy
两者均适用	adjacent_difference copy merge partial_sum remove_copy remove_copy_if replace_copy set_difference set_intersection set_symmetric_difference set_union transform unique_copy

因为 `istream_iterator` 用作 `input_iterator`，`ostream_iterator` 用作 `output_iterator`，因而它们可以使用这些算法。程序清单 7-4 中的程序使用了带有一个输出迭代器的 `transform()` 算法。当然，当在进程间通信情况下应用任何这样的算法时必须小心谨慎。

7.4 使用接口类来实现面向对象命名管道

用于实现面向对象匿名管道的技术有两个弊端。首先，文件描述符没有封装，这意味着它可以被程序中的任何函数操纵。其次，通过匿名管道可以实现的进程间通信只限制于关联进程。也就是说，只有父进程和子进程可以使用匿名管道来执行进程间通信。我们可以使用更强大的管道

类型——命名管道 (named pipe) 来解决这两个问题。

要讨论命名管道, 我们必须扩展管道的抽象概念。以前我们已经讲过, 管道必须具备 7 个基本组件:

- 输入端口;
- 输出端口;
- 数据缓冲器;
- 数据缓冲器插入操作;
- 数据缓冲器提取操作;
- 数据缓冲器创建操作;
- 数据缓冲器析构操作。

命名管道为以上组件添加了新的概念和组件。命名管道为管道概念添加的最重要特征是, 它们允许不关联的进程通过文件名的等价体来访问一个管道。

7.4.1 相关客户/服务器术语

当进程使用匿名管道进程间通信时, 进程是关联的。对应进程或者是父进程, 或者是子进程。命名管道的概念改变了这种术语结构。通过名字打开管道的进程称做客户 (机) 进程 (client process)。服务器进程 (server process) 可以有多个使用管道的客户进程。服务器进程负责建立命名管道的属性。

7.4.2 名字包含哪些内容

创建命名管道时, 它们被赋予了某文件名的等价名字。任何知道这个管道名字而且具有所需访问权限的进程都可以打开、读取和写入管道。在 Win32 和 OS/2 环境中, 命名管道的强大功能之一是管道的名字可以包含服务器或网络上另一台机器的域名。例如, 我们可以按如下方式命名管道:

```
\\ServerName\PIPE\mypipe
```

管道的名字是 mypipe。不过, 这个名字同时指定 ServerName 作为 mypipe 位置的一部分。ServerName 可以指另一台远程计算机。例如, 如果在计算机 ResearchGroup1 上有一个进程打开了 \\ServerName\PIPE\mypipe, 我们就要通过网络在 ResearchGroup1 和 ServerName 之间进行进程间通信。自然执行这些进程所在的网络也必须运行命名管道。

所以, 必须给管道抽象添加的第一个组件是一个名字。在 UNIX 环境中, 命名管道有时称做 FIFO, 它在系统中有一个文件名, 而且是一个永久结构。FIFO 是一种单向结构。也就是说, 在 UNIX 环境中, 命名管道的用户必须打开它进行读或写, 但不能既读又写。在 UNIX 环境中创建的命名管道一直保留在文件系统中, 除非显式从程序中使用 unlink() 删除它们, 或使用一些命令提示符命令来删除它们, 如 rm 命令。

在 UNIX 和 Win32 环境中, 命名管道有一套相关权限。如果管道用户不具备适当的权限 (UNIX) 或安全属性 (Win32), 就不允许访问管道。因此, 我们还要给管道概念添加另一个组件, 这就是访问权限 (access right)。虽然 UNIX 的命名管道概念与匿名管道非常相似, 但在 Win32 和

OS/2 环境中,命名管道相对于匿名管道来说具有巨大的优势。在 UNIX 环境中,无关联进程可以使用命名管道。不过,这些进程必须位于同一台计算机上。在 Win32 和 OS/2 环境中,命名管道可以通过网络来访问。无关联和有关联进程都可以使用命名管道,而且可用于实现网络环境下的客户服务器模型。命名管道支持客户与服务器间的多对多(many-to-many)、一对多(one-to-many)、多对一(many-to-one)以及一对一(one-to-one)关系。

在 Win32 和 OS/2 环境中,命名管道可以按消息或字节模式打开。在消息模式中,读取和写入管道的信息被看作消息。消息至少由两部分组成:消息头和消息体。在字节模式中,读取和写入管道的信息被看作匿名字节序列。命名管道可以按输入管道、输出管道或既是输入管道又是输出管道的方式打开。因此,我们可以给管道概念再添加两个组件:打开模式(open mode)和管道模式(pipe mode)。打开模式决定管道是否用于输入、输出,或既输入又输出。管道模式决定管道是否为消息流或字节流。打开模式还告诉我们管道的读取和写入是否处于阻塞或非阻塞模式。输入端口可能与输出端口大小不同,所以我们还必须添加输入端口大小和输出端口大小。通过抽象继承机制,基本命名管道的最终抽象至少具备以下组件:

- 管道名字;
- 访问权限;
- 打开模式;
- 管道模式;
- 输入端口;
- 输出端口;
- 输入端口大小;
- 输出端口大小;
- 数据缓冲器;
- 数据缓冲器插入操作;
- 数据缓冲器提取操作;
- 数据缓冲器创建操作;
- 数据缓冲器析构操作。

虽然在 UNIX 环境中命名管道不要求具备所有的组件,但在 Win32 和 OS/2 环境中,这些组件却是命名管道的最低要求。如果在 UNIX 环境中需要与 Win32 和 OS/2 环境中相似的功能,则应当使用套接字(socket)。UNIX 套接字允许通过企业内联网和国际互联网进行无关进程间的通信。

找到抽象命名管道具备的基本组件后,我们就可以使用类结构来为管道建模。使用类结构,我们还封装创建管道的文件描述符。这意味着只有命名管道类的成员才能访问文件描述符。当应用程序需要 IPC 时,封装有助我们编写更安全的多线程应用程序。除了封装文件描述符外,我们还在命名管道对象和 fstream 类家族之间建立了重要的连接。

7.4.3 命名管道和 iostream 复合

不是简单地将文件描述符附加到 iostream 对象上(就像使用匿名管道一样),下面我们将在构建命名管道对象包含 Pstream 类期间使用复合(composition)。图 7-8 所示的命名管道类关系图就

显示了 `fstream` 类与 `npstream` 类间的关系。

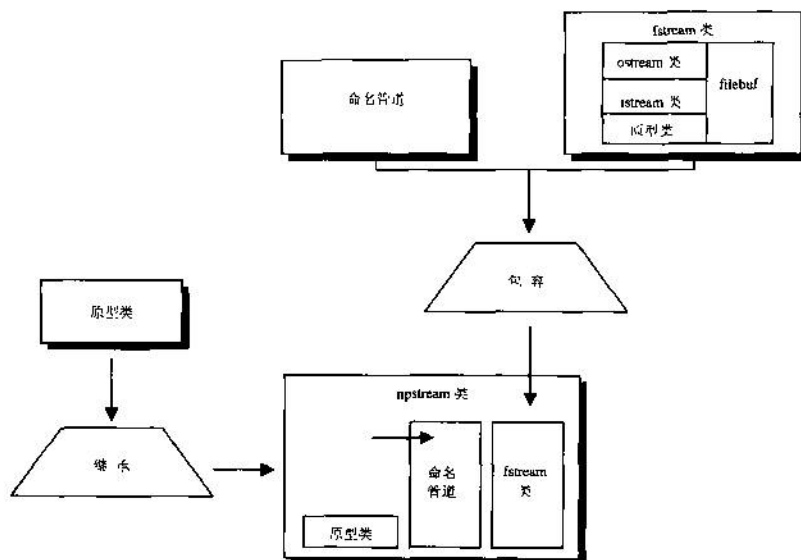


图 7-8 使用命名管道对象的类关系图。该图显示了 `fstream` 类和 `npstream` 类之间的关系。

`npstream` 类用作命名管道提供系统 API 面向对象接口的封装器

`npstream` 类是接口类的另一个例子。我们使用这个类作为给命名管道提供系统 API 面向对象接口的封装器。通过与 `fstream` 对象的包容关系（containment relationship），`npstream` 类是命名管道功能与 `iostream` 提供的 I/O 面向对象模型的合成。这意味着我们可以使用 `npstream` 类来创建用于关联进程间、非关联进程间以及网络间通信的对象。图 7-9 显示了 `npstream` 类、`iostream` 类、命名管道、插入器、提取器以及单独进程间的关系。

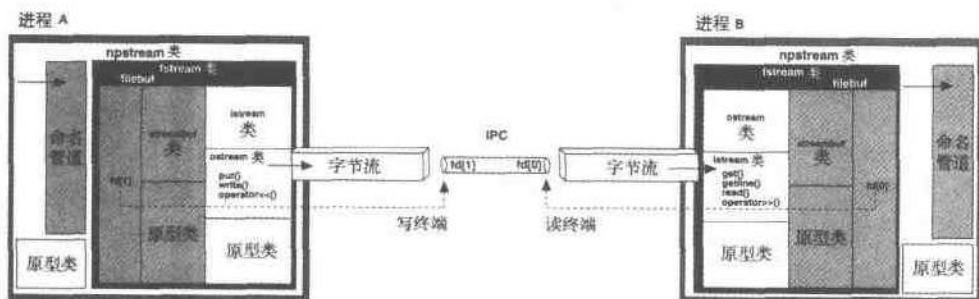


图 7-9 `npstream` 类、`iostream` 类、命名管道、插入器、提取器以及单独进程间的关系

7.4.4 `npstream` 接口类

下面让我们详细讲解如何构建 `npstream` 类。`npstream` 类的声明如下所示：

```

template<class T> class npstream{
private:
    char PipeName[20];
    unsigned long PipeHandle;
    long in PipeMode;
    long in OpenMode;
    long in OutputBufSize;
    long in InputBufsize;
    long in TimeOut;
    unsigned long Result;
    fstream NamedPipe;

public:
    npstream(char *PName,long PMode,
              long OMode,long TOut=1000);
    npstream(char *PName);
    ostream &operator<<(T &X);
    istream &operator>>(T &X);
    long in pipeMode(void);
    long in openMode(void);
    long in outBufSize(void);
    long in timeOut(void);
    unsigned long result(void);
};

```

这个类构建为一个模板类，所以我们通过任何种类的对象来使用这个命名管道。此对象的唯一要求是定义了插入器<<和提取器>>运算符。我们将数据成员声明为私有（private）。这个类不是作为一个节点类，而且不是为继承而设计。大部分 IPC 类都是具体类。一般而言，接口类不应当作节点类。

一、接口类中的完全封装

一般经验是，当 C++ 类用作特定操作系统服务的接口类时，这个类应当封装与该服务相关的每个函数和所有数据。如果这个类选择不封装每个函数，它至少必须对系统服务提供的每个函数配备一个等价体。该类或类的家族应当能完全表示它所封装的操作服务。用户应当能够从接口类中得到完整的功能。否则，接口类就没有提供完整的接口。npstream 类封装了 Win32 或 OS/2 调用 CreateNamePipe() 或 DosCreateNPipe() 函数所需的所有参数。npstream 类最终封装了与命名管道相关的所有系统服务，或者提供了等价体。NamePipe fstream 对象将包含与命名管道的实际连接。如何实现呢？图 7-10 剖析了 fstream 类的类关系。

请注意，fstream 类包含一个 filebuf 组件。filebuf 组件是通过文件描述符附加到命名管道上的组件。npstream 类有两个构造函数：

```

template<class T> npstream<T>::npstream(char *PName,
                                         long PMode,
                                         long OMode,
                                         long TOut=1000)

```



```

{
    strcpy(PipeName, PName);
    PipeMode= PMode;
    OpenMode=OMode;
    OutputBufSize=(sizeof(T)*1000);
    InputBufSize=(sizeof(T)*1000);
    TimeOut=TOut;
    Result=DosCreateNPipe(PipeName,&PipeHandle,OpenMode,
                          PipeMode,OutputBufSize,
                          InputBufSize, TimeOut);
    DosConnectNPipe(PipeHandle);
    NamedPipe.attach(PipeHandle);
}

template<class T>npstream<T>::npstream(char *PName)
{
    DosOpen(PName, &PipeHandle,&Result,0,
            FILE_NORMAL,FILE_OPEN,
            OPEN_ACCESS_READWRITE,
            IOOPEN_SHARE_DENYNONE,
            (PEAOP)NULL);
    NamedPipe.attach(&PipeHandle);
}

```

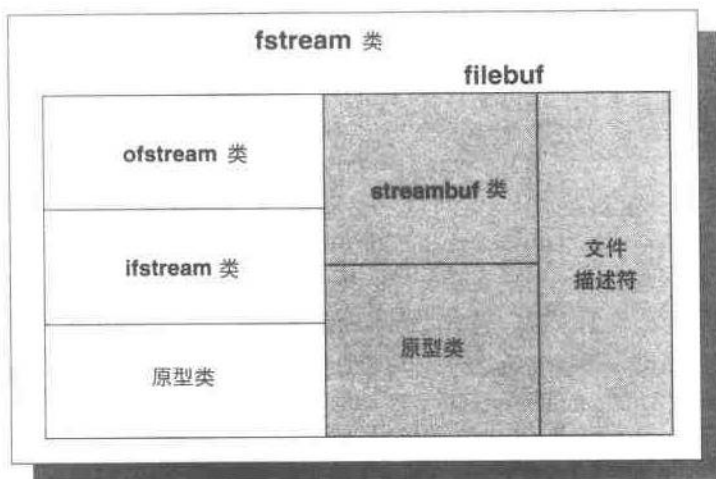


图 7-10 fstream 类的类关系剖析

服务器进程使用第一个构造函数构建命名管道。这个构造函数包含对 `DosCreateNPipe()` 的实际调用。虽然这些例子包含 OS/2 系统 API 调用，但 Win32 调用实际上是相同的，而且功能也是一致的。首先创建管道。变量 `PName` 包含管道的名字。如果名字还包含网络服务器的对应名字，

则通信发生在远程进程间。创建管道后,使用 `DosConnectNPipe()` 函数将它设置成监听模式(listening mode)。一旦将管道放入监听模式后,客户进程就可以打开这个管道进行读和写。将管道放入监听模式后,通过调用 `attach()` 成员函数建立与 `fstream` 对象 `NamedPipe` 的连接。

```
NamedPipe.attach(PipeHandle);
```

`attach()` 成员函数接受与命名管道相关的描述符。一旦将管道附加到 `fstream` 对象,使用 `iostream` 类的诸多优点立即显现。例如,执行与管道之间的自动翻译。程序员可以充分利用 `iostream` 的 `ios` 组件的格式和状态功能。例如:

```
template<class T>ostream &npstream<T>::operator<<(T &X)
{
    NamedPipe<<X<<" ";
    return(NamedPipe);
}
```

成员函数用于在命名管道中插入对象。可以对每个定义了运算符<<的对象进行这个操作。当定义了这个运算符后,程序员不必担心类型转换。该转换由 `istream` 或 `ostream` 组件完成。插入器和提取器使管道读取和写入更简单。

客户进程使用第二个构造函数打开与管道的连接。请注意,第二个构造函数没有调用 `DosCreateNPipe()` 函数,而是通过传统的打开调用来打开命名管道:

```
DosOpen(PName, &PipeHandle,&Result,0,
        FILE_NORMAL,FILE_OPEN,
        OPEN_ACCESS_READWRITE
        TOPEN_SHARE_DENYNONE,
        (PEAOP) NULL);
```

一旦客户端打开了管道, `fstream` 对象使用 `attach()` 成员函数连接到该管道。建立连接后,无论什么时候客户与 `fstream` 对象 `NamedPipe` 交互,客户实际上是与连接到命名管道的 `filebuf` 进行交互。因为 `NamedPipe` 对象和 `PipeHandle` 被封装在 `npstream` 类中,所以我们在多线程或多任务环境中进行的任何管道交互更安全。虽然在 C++ 中,可以不通过封装来解决,但难以顺利访问受保护或私有数据成员或成员函数。这种保护通常是防止数据竞争条件所必需的。

程序清单 7-5 是一个服务器进程,它创建了一个命名管道,从客户进程中读取一个字符串,然后显示该字符串

程序清单 7-5 一个服务器进程,它创建了一个命名管道,使用>>运算符从客户进程中读取一个字符串,然后显示该字符串

```
#define INCL_DOSPROCESS
#include "npipes2.cpp"

void main(void)
{

    long int OMode = NP_ACCESS_DUPLEX;
    long int PMode = NP_UNLIMITED_INSTANCES;
    char X[6] = "";
```

```

    npstream<char *> Server("\\pipe\\mypipe", PMode, OMode);
    Server >> X;
    X[5] = '\\0';
    cout << "From Client " << X << endl;

}

```

请注意，读取管道是使用运算符>>来完成的，这一点与调用系统 API 相反。这种语义保持了 `iostream` I/O 模型的信号量。将看作匿名字节序列的数据插入输出流中，或从输入流中提取。在本例的命名管道中，流正是进程间的通信渠道。还请注意，对管道文件描述符的访问不再那么方便了。程序清单 7-6 中的程序演示了一个在命名管道中写入一个字符串的客户进程。

程序清单 7-6 在命名管道中写入一个字符串的客户进程

```

#include "npipes2.cpp"

void main(void)
{
    char Message[] = "HELLO";
    npstream<char *> Client("\\pipe\\mypipe");
    Message[5] = '\\0';
    Client << Message;
    cout << "To Server X = " << Message << endl;
}

```

为了让客户与服务使用相同的管道，客户必须在构造函数中指定正确的名字。在这里，该名字就是 `mypipe`。

二、`npstream` 状态

`npstream` 对象有两种与之关联的状态。第一种状态包括管道状态以及附加到管道上的 `fstream` 对象状态。这种状态只对于 `npstream` 类的成员可用，主要是因为 `fstream` 对象和状态变量是私有的。`fstream` 对象的状态是由它的 `ios` 组件决定，它封装了三种状态：

- 打开模型状态；
- 缓冲器状态；
- 格式状态。

打开模型状态

流的打开模型状态显示文件是否打开用于读、写，或者既读又写。这个状态还显示流是否按文本或二进制模式打开。

缓冲器状态

对于 `NamedPipe` `fstream` 对象，缓冲器状态包含在一个 `filebuf` 对象中。缓冲器状态可以由以下成员函数决定：

```
NamedPipe.goog()
```

```
NamedPipe.bad()
NamedPipe.fail()
NamedPipe.eof()
```

这些成员函数可用于协助决定附加管道是否仍然处于工作条件。它们的意义与在常规文件流中的用法一致。

格式状态

格式状态显示数值精确度、十六进制、十进制或八进制格式的信息。格式状态显示数值插入流中时是左对齐还是右对齐的信息。格式状态还显示数值是定点、浮点，还是科学记数法的信息。例如，我们可以使用 `ios` 组件的格式状态成员函数来格式插入管道或从管道中提取的信息。这种格式可以使用 `ios` 成员函数或操纵函数来控制：

```
NamedPipe.setf(ios::fixed|ios::showpoint);
NamedPipe.precision(4);
NamedPipe<<setiosflags(ios::hex);
```

`npstream` 对象的其它状态变量指定了：

- 管道模式；
- 打开模式；
- 管道名字；
- 实例计数；
- 输入缓冲器大小；
- 输出缓冲器大小；
- 时间耗尽。

`npstream` 类的客户可以访问第二种状态。这种状态由以下成员函数决定：

```
long int pipeMode(void);
long int openMode(void);
long int outBufsize(void);
long int inBufsize(void);
long int timeOut(void);
```

`npstream` 类首先查看我们使用接口类封装系统服务的方式。`npstream` 类保护文件描述符，并通过连接到 `iostream` 提高命名管道的性能。使用 `iostream`，我们可以利用 `filebuf` 和 `ios` 的功能。

7.4.5 命名管道与 STL `istream_iterator` 和 `ostream_iterator`

与匿名管道可以连接到 `istream_iterator` 和 `ostream_iterator` 一样，命名管道也可以通过 `iostream` 对象进行连接。扩展 `npstream` 类定义另一个运算符 `<<`：

```
void operator<<(vector<T>&&X);
```

这个重载后的运算符 `<<` 将接受一个任何数量 `T` 类型对象的矢量容器，而且使用一个与 `NamedPipe fstream` 对象的连接来实例化名叫 `Out` 的 `ostream_iterator` 对象。然后，它使用 STL `copy()` 算法将矢量对象中的所有元素写入管道。运算符定义如下：

```
template<class T>void npstream<T>::operator<<(vector<T> &X)
{
    ostream_iterator<T> Out(NamePipe, " ");
```

```

        copy(X.begin(), X.end(), Out);           //沿管道发送对象
    }

```

图 7-11 显示了 `npstream` 类、`ostream_iterator`、`fstream` 类、插入器、提取器、命名管道以及进程间的关系方块图

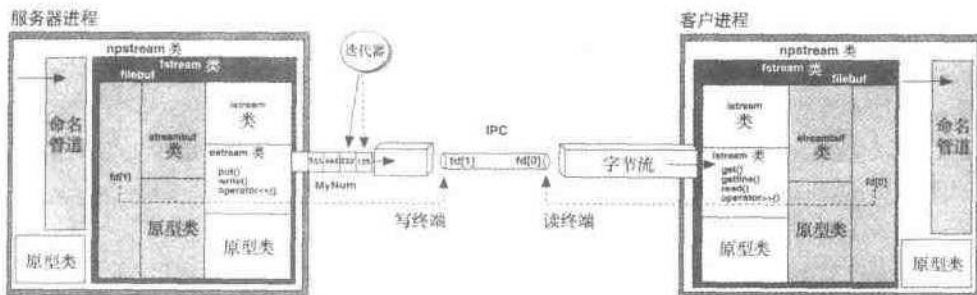


图 7-11 `npstream` 类、`ostream_iterator`、`fstream` 类、插入器、提取器、命名管道以及进程间的关系方块图

当面向对象 IPC 组件用于进程间通信时，产生的应用架构更安全，而且控制性更强。程序清单 7-7 中的服务器进程演示了 `ostream_iterator` 与管道的使用。

程序清单 7-7 服务器进程声明了一个 `int` 类型矢量，其中有 4 个 `int` 插入到此矢量。程序还创建了一个命名管道，将矢量沿管道发送。程序中应用了 `<<` 运算符

```

#define INCL_DOSPROCESS
#include "npipe2.cpp"

vector<int> MyNum;

void main(void)
{
    long int OMode = NP_ACCESS_DUPLEX;
    long int PMode = NP_UNLIMITED_INSTANCES;
    npstream<int> Server("\\pipe\\mypipe", PMode, OMode);
    MyNum.push_back(125);
    MyNum.push_back(222);
    MyNum.push_back(444);
    MyNum.push_back(555);
    Server << MyNum.size();
    Server << MyNum;
    cout << "Server Done";
}

```

程序清单 7-7 中的服务器进程声明了一个 `int` 矢量，名字为 `MyNum`。在 `MyNum` 中插入 4 个 `int`。服务器进程创建一个命名管道，并沿着管道发送矢量 `MyNum`，代码如下：

```
Server << MyNum.size();
```

```
Server << MyNum;
```

服务器首先发送写入管道的项数量。接着通过管道发送矢量。重载运算符<<导致管道处理是透明的,而且使在命名管道中写入元素列表更简单。程序清单 7-8 中的客户程序读取管道,并显示从服务器发送的 int 列表。

程序清单 7-8 读取管道,并显示从服务器发送的 int 列表的客户进程

```
#include "npipes2.cpp"
void main(void)
{
    int Size;
    int N;
    int Value;
    npstream<int> Client("\\pipe\\mypipe");
    Client >> Size;
    for(N = 0; N < Size; N++)
    {
        Client >> Value;
        cout << "From Server " << Value << endl;
    }
}
```

下面是以上客户程序的输出结果:

```
From Server 125
From Server 222
From Server 444
From Server 555
Server Done
```

iostream、STL 迭代器以及管道之间的连接有许多有用的应用。大部分可与 istream_iterator 或 ostream_iterator 使用的 STL 算法也可以通过 iostream 连接应用于命名管道。这也意味着 iostream 和 STL 容器可用于网络上的进程间通信。显然,为了让 npstream 类用于复制,必须添加异常处理。事件互斥量和其它同步变量应当用于协调管道的应用。创建管道的构造函数应当支持多个客户的多线程处理。必须建立一个管道断开策略等等。本章所列举的程序清单 7-5 到 7-8 只是指导性地为程序员演示了实现封装进程间通信系统 API 的接口类的一些可能情况。

同步对象

一个保持一定流体稳定性的系统可能由于毫厘之差而被破坏。系统有次序，从一点流到另一点。如果有东西阻碍了流体，次序就不复存在。不受控制者可能会避免这种次序的消失。这就是为什么生态的最高功能是理解其因果关系的原因。

Dune

——Pardot kynes, First Planetologist of Arrakis

使用类与系统服务的一个优点是类可以为这些服务提供系统独立的接口。在第7章，我们已经看到了这样一个例子，其中的接口类为匿名管道和命名管道服务提供一个面向对象接口。提供面向对象接口不仅仅是提供一个新接口，这也是优点之一。通过封装、继承和多态，同步变量更安全，而且可预测性更强。通常而言，同步变量（如互斥量）和条件变量声明为全局，所以，多个线程可以访问它们。虽然这为需要使用同步变量的所有线程提供了方便，但它没有提供这些线程遵循的策略。这意味着线程可以按任意方式自由使用（或误用）同步变量。而且，传统同步变量没有为程序员提供实际的方法，将变量与需要同步的数据或设备直接关联。这意味着同步变量可能无意间被混合和匹配以同步化任何数据块或设备。

易于声明同步变量既是好消息，也是坏消息。好消息是程序员可以轻易地在程序中添加同步技术。坏消息是程序员对同步变量的使用提示或原则一无所知。这通常会导致多线程处理的混乱。在小型或简单的程序中使用自由漂浮的全局同步变量也是相当复杂的。当把同样的技术应用于中型或大型多模块或者多人开发工作中时，这种混乱更显而易见，通常包括：

- 不可捉摸的漏洞；
- 不可预测的行为；
- 不正确的结果；
- 变相竞争条件；
- 不可维护的代码；

- 不可重用的代码;
- 莫名其妙的性能下降;
- 脆弱的系统架构;
- 不可恢复的应用错误;
- 死锁;
- 无限延迟。

随着软件工程变得复杂,单个程序中源模块的数量逐渐增加。在大型多线程程序中,同步变量可能在一个源模块中全局性声明,程序内的众多源模块都可以访问。跟踪哪一个源模块正锁定或取消锁定哪一个同步变量是一项繁琐而且容易出错的任务。而且,如果同步变量的数量很大,分清哪些变量与哪些数据对应使用变得更困难。

Bertrand Meyer 在他的 *Object-Oriented Software Construction* 一书中,详细讨论了相对于使用过程化途径开发的软件来说,用面向对象编程技术能开发出更可靠、可重用性更强的软件的重要原因。在“面向对象之路”一章中,他描述了为什么真实系统没有最高层,为什么数据必须是宇宙的中心。封装、继承和方法访问是根治受困于大小和复杂性的软件技术的唯一处方。

使用面向对象技术开发多线程应用为并发提供了最可靠的途径之一。

我们在两个前端应用面向对象技术去同步化变量。首先,互斥量或条件变量以及所有的相关系统服务都被封装在一个接口类中。这意味着互斥量对线程内的所有函数或过程不再具有全局性。对互斥量的唯一的访问途径是通过类成员函数。其次,接口类与即将通过继承或复合同步的数据或设备直接关联。因为互斥量将被嵌入它所保护的数据或设备中,所以不存在保护哪一个互斥量的问题。同步变量可以用于调节访问同一个临界区的多个线程。同步变量也可以用于调节访问同一个临界区的多个进程。让我们首先看看可以使用 C++ 结构构建的 `mutex` 类,而且用来调节访问临界区的多线程。

8.1 初识 `mutex` 类

用于保护线程内临界区的相互排斥(即互斥, `mutex`) 变量有多种类型。最简单的 `mutex` 类型在某一时刻只能由一个线程占有。线程试图锁定该互斥量。如果互斥量没有被占有,线程将获得互斥量的所有权,并能够访问互斥量保护的数据或设备。如果互斥量已经被占有或锁定,请求互斥量的线程将被阻塞,而且直到取消锁定请求的互斥量前一直处于阻塞。最简单的互斥量只支持少数操作:

- 创建;
- 锁定(被占有);
- 取消锁定(被释放)
- 销毁。

这些操作非常适合于 C++ 类结构。互斥量创建的任务可以分配给类构造函数来完成。可分配互斥量的类析构函数来析构(destruction)。锁定和取消锁定操作可以分配给类成员函数。我们可以使用这些类构造函数来实现 `mutex` 类。另外一个要考虑的问题是,如果由于某些原因,互斥量

操作失败了，将会发生什么呢？例如，构造函数不允许返回值。类的用户如何才能知道该互斥量是否正确创建了呢？这个问题对于类析构函数的互斥量也存在。锁定或等待其它线程的互斥量不应当销毁。如果析构函数试图销毁这样的互斥量，那么类的用户如何才能知道呢？所以在类的设计中，我们必须提前决定万一错误条件发生时应当采取的措施。这是必需的步骤，因为 `mutex` 类被嵌入它保护的类中。受保护类的用户不必关心嵌入同步变量的错误状态检查。一定能找到某种途径，让互斥量的客户确信互斥量处于可接受状态。

此时，面向对象异常处理就发挥作用了。在多线程架构中，面向对象异常处理用于简化错误的不期望程序行为的处理过程。为了解决 `mutex` 类中错误条件的问题，我们将使用 C++ 的异常处理功能。我们现在知道用哪一个 C++ 结构来创建 `mutex` 类，我们需要 C++ 结构将访问的系统服务。这里的第一个 `mutex` 类，我们将使用 POSIX 互斥量。表 8-1 列出了我们将在 `mutex` 类中使用的基本互斥量 API。

表 8-1 在 `mutex` 类中使用的基本互斥量 API

针对互斥量的 POSIX API	描 述
<code>pthread_mutex_init()</code>	初始化互斥量
<code>pthread_mutex_lock()</code>	获取互斥量的锁定。阻塞其它试图锁定同一个互斥量的线程
<code>pthread_mutex_unlock()</code>	互斥量的占有才使用它取消锁定该互斥量。如果互斥量没有被占有，它返回一个错误
<code>pthread_mutex_trylock()</code>	如果互斥量没有被锁定，则获得一个锁定，如果它已经被锁定，则返回一个错误
<code>pthread_mutex_destroy()</code>	销毁一个初始化的互斥量

表 8-1 中的 POSIX API 对应于互斥量的三种类型操作。API 用于互斥量的创建、锁定和析构。图 8-1 中的类关系图显示了 `mutex` 类的结构。

请注意，`mutex` 类包含一个对象 `general_exception`。这个对象是一个受保护数据成员，因此仅能被类 `mutex` 的成员访问。如果 `mutex` 类在互斥量创建或互斥量析构中碰到问题，就使用 `general_exception` 对象。下面是 `mutex` 类的声明：

```
//声明一个简单 mutex 类
#include<eclases.h>
#include<pthread.h>

class mutex{
protected:
    pthread_mutex_t Mutex;
    general_exception SemException;
public:
    mutex(void);
    ~mutex(void);
```

```

    void lock(void);
    void unlock(void);
};

```

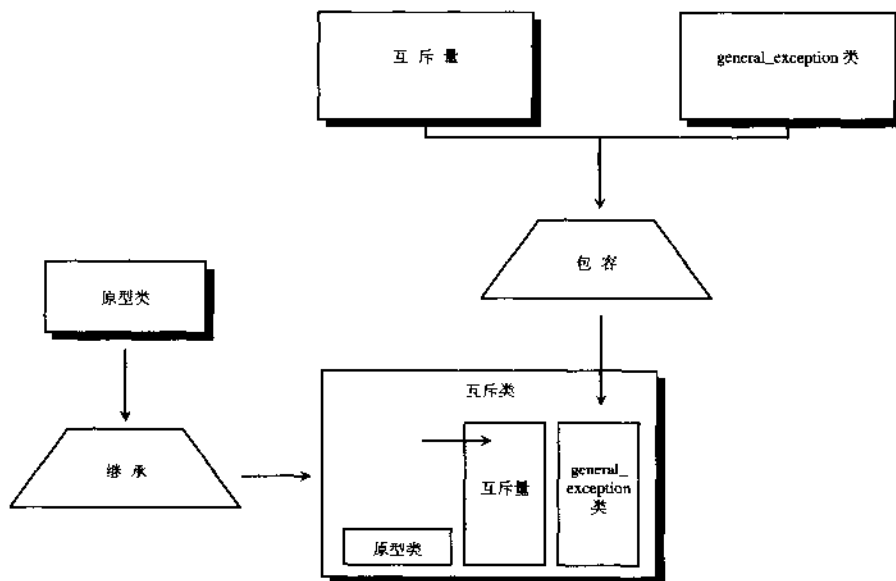


图 8-1 简单互斥量的类关系图

除了 `general_exception` 组件外，`mutex` 类还封装了一个 POSIX `pthread_mutex_t`。通常，这个互斥量将在一个适当大小的作用域内声明，以便于需要访问的任何线程都可以获得锁定。因为 `Mutex` 被封装在 `mutex` 类中，所以只有该类的成员可以访问它。因为我们只关心最简单的互斥量，所以，构造函数不带有任何参数。下面是构造函数的定义：

```

mutex::mutex(void)
{
    if(pthread_mutex_init(&Mutex,NULL)){
        SemException.message("Could Not Create Mutex");
        throw SemException;
    }
}

```

构造函数调用 POSIX `pthread_mutex_init()` 函数。如果构造函数不能初始化互斥量，它将抛出一个异常。互斥量创建可能由于多种原因而失败，一个常见的原因是到达了系统的互斥量极限。另一个原因是，程序员可能请求互斥量一个在特定环境中不可用的属性。例如，程序员可能请求创建一个被进程共享的互斥量。如果运行时环境不支持这种互斥量，互斥量创建就不会成功。如果互斥量创建失败，抛出异常很重要。如果我们不抛出异常，也可以设置某种标志，并要求类的用户检查该标志的值。此外，检查错误标志在简单或小型程序中工作良好。但在中型或大型 C++ 程序中，错误处理策略不应依赖于用户检查错误标志。我们将在后面内容中详细讨论异常处理（exception handling）。在这里，要注意的重要的一点是，由于 C++ 构造函数不支持返回值，所以

我们使用了异常处理。

SemException 对象是 general_exception 类型。general_exception 类是一个常规 C++ 类。我们将在本书的后面讲解中使用这个类。因为 general_exception 是一个常规类，所以我们可以使用继承和多态来将它特殊化，让我们设计特定上下文的异常类。类 general_exception 在文件 eclases.h 中声明，eclases.h 位于本书附带磁盘的程序目录中。下面是 general_exception 的声明：

```
class general_exception{
protected:
    char Operation[2];
    char Message[81];
public:
    general_exception(void);
    general_exception(char *Msg);
    general_exception(const general_exception &N);
    general_exception &operator=(const general_exception &N):
    void operation(char *Op);
    char *operation(void);
    void message(char *Msg);
    char *message(void);
};
```

使用 general_exception 类，我们赋予异常一条特殊的消息，描述出错的内容。我们也可以捕获导致异常的特定操作。通过在异常发生时指定描述信息，处理抛出异常的处理器可以访问这些有意义的诊断信息。在面向对象编程中这一点很重要，因为异常条件可能发生在对象层次的深处，或者由于错误重载运算符而产生。当构建多线程面向对象架构时，异常的使用是必需的。在 mutex 类的构造函数中，我们使用如下代码：

```
SemException.message("Could Not Create Mutex");
```

来捕获错误条件。不过，如果互斥量创建成功，mutex 类的任何占有者都可以请求互斥量的占有权。通过调用 lock() 成员函数来实现这一点。使用 mutex 类与传统互斥量方法的重要区别在于互斥量类型的对象不会声明为自由漂浮或全局的对象。互斥量类型对象设计用作数据类或设备类的组件。互斥量可以私有性地被一个数据或设备类继承。mutex 类也可以嵌入另一个使用复合的类中。

当 mutex 类仅作为另一个类的组件被访问时，就不存在保护哪一个互斥量的问题。它保护它所属的类部分。互斥量保护该类具有的所有临界区。包含互斥量的类决定部署互斥量的方式和时间。在对象中嵌入互斥量决定对象在多线程环境中使用的方式。对象的用户不必担心同步化对象的使用，因为对象同步化自身。同步临界区的责任从对象的用户转而由对象的生产者承担。在第 1 章中，我们说过抽象数据类型既定义它的数据组件，同时定义对数据组件的合法操作。因为类调控它的数据组件的访问权，在并行环境中保护数据组件的责任就落在类身上。这是与面向对象的基本概念相一致的。例如，我们可以使用 mutex 类来保护对用户自定义类 mt_rational 的访问。mt_rational 的声明如下所示：

```
#include <rational.h>
#include <ctmutex.h>
```

```

class mt_rational:public rational{
protected:
    mutex Mutex;
public:
    mt_rational(long Num=0,long Den=1);
    mt_rational(const mt_rational &X);
    mt_rational &operator=(const mt_rational &X);
    void assign(long X, long Y);
};

```

mt_rational 类封装了一个互斥量类型的 Mutex 对象。Mutex 对象将用于保护 mt_rational 的临界区。我们设计了两个临界区。第一个临界区在用户调用 assign() 成员函数给 mt_rational 类型对象赋值时发生。第二个临界区在赋值语句用于给 mt_rational 类型对象赋值时发生。mt_rational 的类关系如图 8-2 所示。

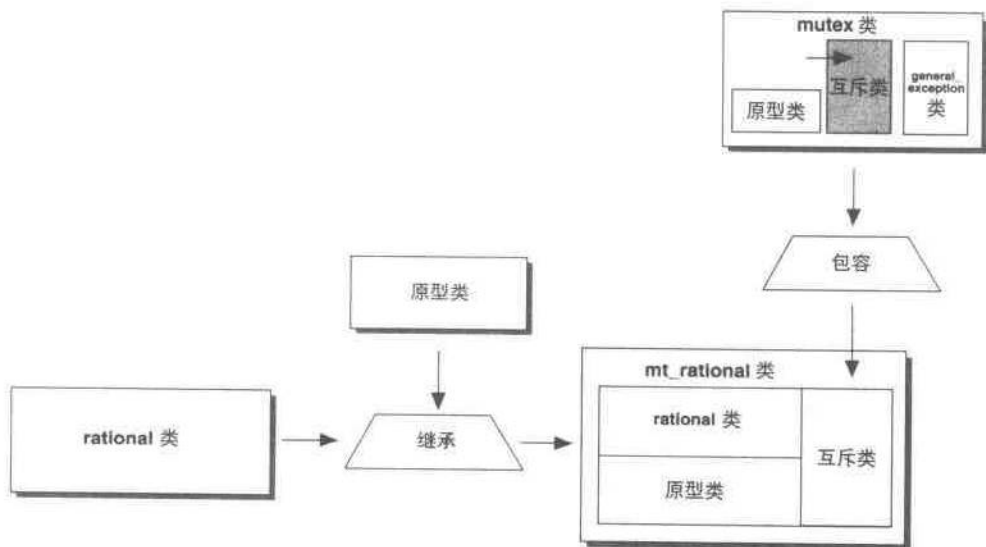


图 8-2 对于 mt_rational 类的类关系

图 8-2 显示类 mt_rational 继承类 rational，而且与类 mutex 具有包含关系。当混合类层次与同步和非同步类时必须小心。当一些派生类具有锁定，一些派生类不具有锁定时，运行时多态可能变得特别复杂。如果使用对基类的引用，得到的结果可能并不是所期望的。第 8 章将探讨如何应用锁定的技术。使用 Mutex 类的两个成员函数是 assign() 和 operator=()。下面是这些成员函数的声明：

```

void mt_rational::assign(long X, long Y)
{
    if(Y==0){

```

```

        Exception.message("Zero Is A Invalid Denominator");
        throw Exception;
    }
    try{
        Mutex.lock();
        Numerator=X;
        Denominator=Y;
        Mutex.unlock();
    }
    catch(general_exception &M)
    {
        Mutex.unlock();
        throw M;
    }
}

mt_rational &mt_rational::operator=(const mt_rational &X)
{
    try{
        Mutex.lock();
        Numerator=X. Numerator;
        Denominator=X. Denominator;
        Mutex.unlock();
    }
    catch(general_exception &M)
    {
        Mutex.unlock();
        throw M;
    }
    return *this;
}

```

因为 assign()成员函数是用来设置 Numerator 和 Denominator 值的函数之一，所以它包含一个临界区。这个临界区发生在如下赋值语句之时：

```

Numerator=X;
Denominator=Y;

```

如果多个线程试图使用同一个 mt_rational 对象的 assign()成员函数，该值可能会被破坏。所以，在允许赋值之前，mt_rational 对象必须能锁定它的互斥量。如果当一个线程访问 mt_rational 对象的 assign()函数时，线程包含另一个 mt_rational 对象调用 assign()函数，无论是哪一个 assign()调用，首先到达 Mutex.lock()者能够访问 Numerator 和 Denominator。另一个线程阻塞于 assign()，直到内部互斥量取消锁定为止。虽然这一简单技术阻止了两个线程同时更改 mt_rational 类型的对象，但 mt_rational 不是完全安全的。一个线程可能在读取 mt_rational，而另一个线程可能正在写 mt_rational。在某些情况下，这是可以接受的。通过 mutex 类封装对 Numerator 和 Denominator 的访问可以提高 mt_rational 的安全性。

显然，这种锁定和取消锁定要花费一定开销。使用锁定方案时存在一个平衡点。有时，是效率与安全性的权衡。如果需要即时访问最新变化的对象，则多线程处理或并发可能不是解决之道。如果所有的读必须绝对保证为最新数据，而且时间不是决定性因素，那么，读和写都可以通过锁定方案来控制。mt_rational 例子显示的一个重要技术是，互斥量必须直接与即将保护的数据或设备关联。这通常会从根本上影响设计和实现的细节。

程序清单 8-1 中的程序演示了一种声明多线程可以访问的全局互斥量以及全局对象的途径。

程序清单 8-1 演示不正确使用全局互斥量来保护对象的两个线程

```

01  #include <iostream.h>
02  #include <pthread.h>
03  #include <mt_rational.h>
04  #include <math.h>
05
06  rational Number;
07  pthread_mutex_t Mutex = PTHREAD_MUTEX_INITIALIZER;
08
09  void *threadA(void *x)
10  {
11      long int Q;
12      try{
13          for(Q =2; Q < 500;Q += 4)
14          {
15              pthread_mutex_lock(&Mutex);
16              Number.assign((Q/2),Q);
17              cout << pthread_self() << " Thread A: "
18                  << Number endl;
19              pthread_mutex_unlock(&Mutex);
20          }
21      }
22      catch(general_exception &M)
23      {
24          cout << pthread_self() << M.message() << endl;
25      }
26  }
27
28  //线程 B 不与保护有理对象 Number 的适当互斥量使用合作
29
30
31  void *threadB(void *x)
32  {
33      long int Y;
34      try{
35          for(Y = 5;Y < 500;Y += 5){
36              Number.assign((Y/5),Y);

```

```

37         cout << pthread_self() << " Thread B "
38         << Number << 1;
39     }
40 }
41 catch(general_exception &M)
42 {
43     cout << pthread_self() << M.message() << endl;
44 }
45 }
46
47
48 void main(void)
49 {
50     pthread_t ThreadA;
51     pthread_t ThreadB;
52     pthread_create(&ThreadA, NULL, threadA, NULL);
53     pthread_create(&ThreadB, NULL, threadB, NULL);
54     pthread_join(ThreadA, NULL);
55     pthread_join(ThreadB, NULL);
56     pthread_mutex_destroy(&Mutex);
57 }
58

```

在演示互斥量传统用法的同时，程序清单 8-1 还显现了按这种方式使用互斥量的一个潜在缺陷。变量 `Number` 是一个 `rational` 类型的对象。它是一个全局变量。`threadA()`和`threadB()`都可以更新它。这意味着`threadA()`和`threadB()`使用`assign()`成员函数的代码区域就是临界区。同步变量`Mutex`也声明为全局。这意味着它可以被进程内的所有线程访问。请注意，`threadA()`使用`pthread_mutex_lock()`，它调用 15 行和 18 行来保护它的临界区。问题在于，`threadB()`并不与之合作。`threadB()`中的临界区不被互斥量调控。这意味着`threadB()`仍然调用`Number.assign()`，即使`threadA()`锁定了这个互斥量也是如此。为了使互斥量的概念在同步访问临界区中 useful，访问同一个临界区的每个线程都必须使用相同的互斥量。在这里，`threadA()`更新 `Number` 对象的同时，`threadB()`也可以更新 `Number` 对象。

这说明了使用传统途径同步使用互斥量的一个弱点。互斥量的使用是高度自发的。没有方便的途径来捆绑互斥量和数据。没有强迫互斥量用于特定临界区的结构。程序员可能使用互斥量无意间忘记保护临界区。请将此途径与程序清单 8-2 中程序所使用的途径进行比较。

程序清单 8-2 演示由两个线程更新的可锁定全局变量 `Number` 的使用

```

1 // 程序清单 8-2
2 // 此程序将演示
3 // 由两个线程更新的可锁定全局变量
4 // Number 的使用
5
6 #include <iostream.h>
7 #include <pthread.h>

```



```
8  #include <mt_rational.h>
9  #include <math.h>
10
11  mt_rational *Number;
12
13  void *threadA(void *X)
14  {
15      long int Q;
16      try{
17          for(Q =2; Q < 500;Q += 4)
18          {
19              Number->assign((Q/2),Q);
20              cout << pthread_self() << " Thread A: "
21                  << *Number endl;
22          }
23      } catch(general_exception &M)
24      {
25          cout << pthread_self() << M.message() << endl;
26      }
27  }
28
29  void *threadB(void *X)
30  {
31      long int Y;
32      try{
33          for(Y = 5;Y < 500;Y += 5){
34              Number->assign((Y/5),Y);
35              cout << pthread_self() << " Thread B "
36                  << *Number << endl;
37          }
38      } catch(general_exception &M){
39          cout << pthread_self() << M.message() << endl;
40      }
41  }
42
43
44  void main(void)
45  {
46      pthread_t ThreadA;
47      pthread_t ThreadB;
48      try{
49          Number = new mt_rational(1,1);
50          pthread_create(&ThreadA,NULL,threadA,NULL);
51          pthread_create(&ThreadB,NULL,threadB,NULL);
```

```

52         pthread_join(ThreadA, NULL);
53         pthread_join(ThreadB, NULL);
54         delete Number;
55     }
56     catch(general_exception &X)
57     {
58         cout << "From main( ) " << X.message() << endl;
59     }
60 }

```

以上程序使用了有一个嵌入互斥量的 `mt_rational` 类。这个互斥量用于保护 `mt_rational` 类中的临界区。程序清单 8-2 中的程序演示多个线程如何访问一个 `mt_rational` 类型的全局对象。注意 `threadA()` 和 `threadB()` 都调用 `assign()` 成员函数。程序清单 8-2 与程序清单 8-1 两个程序的区别在于，用户不负责保护 `Number` 对象，它提供 `assign()` 成员函数内的自我保护。按这种方式，锁定互斥量不再是随意的。每次 `assign()` 成员函数被调用，互斥量要么允许线程继续，要么阻塞线程。同步化责任转移到类的设计者身上，可以使同步策略在类中得到封装。无论在哪里使用 `mt_rational` 类型的对象，将自动强制执行包含在 `assign()` 成员函数和 `operator=()` 中的同步策略。程序清单 8-2 中的程序相对于程序清单 8-1 中的程序的第二个主要优点是，`Mutex` 对象被直接嵌入程序清单 8-2 的 `mt_rational` 对象中。这个互斥量只能被 `Number` 的成员函数访问，而在程序清单 8-1 中，`Mutex` 是一个全局同步变量，它可以任意使用或误用，例如，在 `threadB()` 中该使用它的时候没有使用它。

互斥量的误用可能导致竞争条件，就像程序清单 8-1 中的程序一样。因为 `threadA()` 和 `threadB()` 可能同时访问 `Number`，对象的完整性不能得到保证。互斥量接口类的使用防止了这种竞争条件，因为互斥量与更新 `mt_rational` 类型对象的操作捆绑在一起。

8.1.1 命名互斥量类

命名互斥量 (named mutex) 比匿名互斥量 (anonymous mutex) 更灵活。命名互斥量可以被同一进程内的线程以及不同进程间的线程共享。匿名互斥量用于执行同一进程中线程间的同步。匿名互斥量不与文件名字关联，但命名互斥量与某文件名关联。匿名互斥量不用于执行不同进程间的同步。但是，我们可以使用继承来创建一个可用于多进程间的命名互斥量。新的互斥量在前面所创建简单互斥量的基础上又添加了几个新属性。在命名互斥量类中，我们将使用互斥量持续时间 (mutex duration) 作为一种技术来帮助我们避免死锁。以前我们讲过死锁的 4 个主要必需条件：

1. 进程声明排它性控制需求资源。
2. 进程等待其它资源释放时同时占有资源。
3. 不强制从进程中删除。
4. 存在循环等待条件。

通过赋予命名互斥量决定等待锁定多长时间的能力，我们可以指定一个有限的等待期限。如果互斥量不能在此等待期限内获得锁定，线程取消锁定，并继续执行其它处理。定时有助于防止条件 4 的发生。即使循环等待中涉及多线程，如果阻塞机制具备一个指定的时间耗尽段，它不会

无限如此持续。与持续时间属性一起，我们还要赋予互斥量一个名字属性。名字属性允许多个进程访问互斥量。任何知道互斥量名字的进程都可以打开和锁定它。声明命名互斥量如下所示：

```
class named_mutex:public mutex{
protected:

    char MutexName[81];
    int initiallyOwned;
public:
    named_mutex(void);
    named_mutex(char *MName, int Owned=0);
    unsigned long lockDuration(void);
    void lockDuration(unsigned long Dur);
};
```

named_mutex 类为互斥量添加了两类类型的成员函数：**lockDuration()**成员函数以及另一个构造函数。**lockDuration()**让程序员指定多长时间后阻塞对互斥量的锁定尝试（以毫秒为单位）。例如：

```
named_mutex MyMutex("MutexName");
MyMutex.duration(10000);
MyMutex.lock();
```

```
...
...
...
```

以上代码将导致 **MyMutex** 尝试锁定。如果互斥量已被锁定，线程将阻塞不超过 10 000 毫秒（即 10 秒）。如果互斥量在时间耗尽前可用，则互斥量被锁定。命名互斥量添加的第二个成员函数是一个带有两个参数的构造函数。第一个参数是互斥量的名字，第二个参数决定互斥量最初被创建时是否被占有。虽然命名互斥量或匿名互斥量类可以在类外部使用，但推荐将对应的互斥量嵌入它要保护的类中。在程序清单 8-2 中，我们使用简单匿名互斥量来保护一个用户自定义 **rational** 类。在这里，我们将使用命名互斥量来保护一个 STL 双端队列容器。因为 STL 容器现在不是多线程处理的，所以我们使用一个接口类来为插入和删除操作提供锁定。接口类调整 STL 双端队列的接口允许 **locking()** 的执行。只要调用一个插入成员函数，之前和之后将执行 **lock()**。只要调用一个提取成员函数，之前和之后将执行 **lock()**。锁定处理让双端队列能够被多线程同步使用。

通过继承的命名互斥类

mt_rational 类使用复合来包含匿名互斥类。也就是说，互斥量声明为一个受保护数据成员。受保护数据成员的状态阻止 **mt_rational** 的用户直接操纵这个互斥量。互斥量可以直接被 **mutex** 类或它的任何后代操纵。我们使用私有继承来组合命名互斥量和 **npstream** 类。实际结果与之相同。图 8-3 显示了新 **lqueue** 类的类关系。

lqueue 类的用户将不能直接操纵命名互斥量。因为 **lqueue** 类已经作为私有基类继承了 **named_mutex** 类，**lqueue** 类的成员可以访问 **named_mutex**，但从 **npstream** 派生的类却不能。**lqueue** 类不是用来作为节点类的。当从封装了操作系统服务或其它类的接口类创建类层次时，必须小心谨慎。随着类层次变得越来越复杂，保持封装系统服务的原有意义也变成一项困难的任务。有太多的地方容易以不恰当的方式引用基类和虚函数。所以，我们作为私有基类声明 **named_mutex** 类

来阻止将 `lqueue` 类当成一个基类。`lqueue` 的声明如下所示:

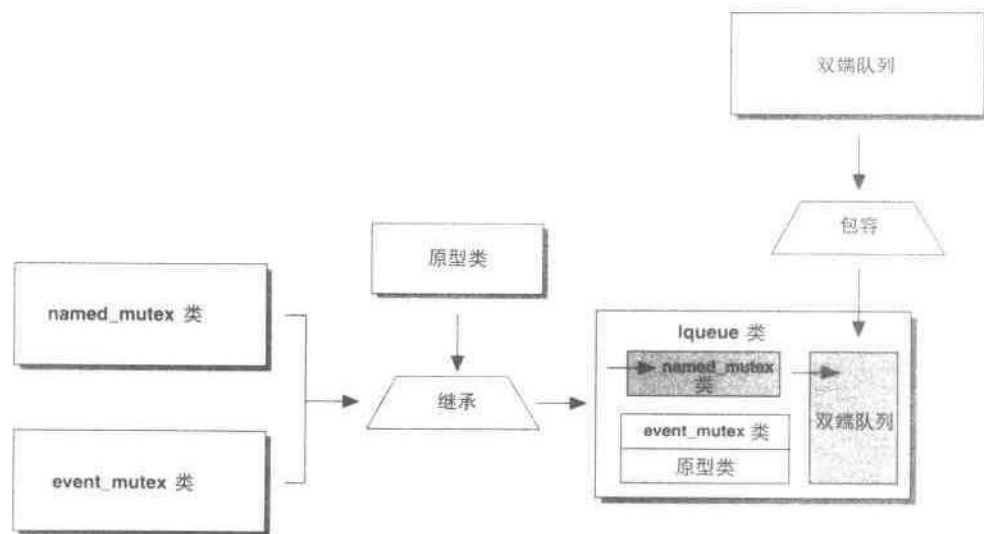


图 8-3 `lqueue` 类的类关系图，它继承了一个命名和事件互斥量

```
#include<deque.h>
#include"ctmutex.h"

template<class T>class lqueue: virtual private named_mutex,
    virtual private event_mutex{
protected:
    deque<T> SafeQueue;
public:
    lqueue(char *MName,int Own,char *EName,int Initial,
        unsigned long Dur);
    inline void insert(T X);
    inline T remove(void);
    inline T front(void);
    inline T back(void);
    inline unsigned int empty(void);
    inline unsigned int size(void);
    inline void erase(void);
    void wait(void);
    void broadcast(void);
};
```

请注意，`deque<T>`类声明为一个受保护数据成员。这意味着这个类的用户不能调用 `deque` 的成员函数。只有 `lqueue` 类封装了成员函数才能被类的用户访问。使用这种技术，程序员只可以输出必需的函数，按需要修改这些函数的接口。在这里，我们没有对受保护 `deque` 类提供迭代器的访问权。迭代器在面向对象多线程环境中被证明是难以管理的。因为迭代器在类的外部，类可能

锁定数据，而且迭代器可能仍然遍历锁定，导致竞争条件。迭代器的灵活性福祸相随。迭代器作为指针抽象的概念允许用户根据需要声明任何数量的迭代器来遍历容器。如果并发执行的多个线程可以使用容器，如何防止竞争条件呢？解决这个问题有两种常见途径。第一种途径是尝试创建一个线程安全（thread-safe）的迭代器。第二种途径是使用传统方法来访问容器或集合类中的对象。在本书中，我们选择了后者。试图创建线程安全迭代器通常会违反面向对象编程的基本思想，所以，我们选择牺牲灵活性来维持面向对象的结构。我们使用单个成员函数来访问集合和容器类中的对象。

在 `lqueue` 类中，`front()`和 `remove()`用于遍历双端队列（`deque`）。使用双端队列来实现队列（`queue`）的概念。所以，数据只能插入 `lqueue` 类的后部，从 `lqueue` 类的前端提取。因为队列是一个 FIFO（先进先出）结构，所以，随机迭代的想法不会实现。FIFO 是一个序列化的列表，其中放入列表中的第一个元素是从列表中删除的第一个元素。放入列表中的第二个元素是从列表中删除的下一个元素，依此类推。

请注意，`named_mutex` 类和 `event_mutex` 类被声明为虚基类。这只允许单个 `named_mutex` 对象和单个 `event_mutex` 对象在发生多继承的事件中被继承。例如：

```
class A: public named_mutex;
class B: public named_mutex;
class C: private A, private B;

class D: virtual public named_mutex;
class E: virtual public named_mutex;
class F: private D, private E;
```

类 C 将包含 `named_mutex` 类型的两个对象，类 F 只包含一个。拥有了与 `lqueue` 类关联的 `named_mutex` 后，我们使用这个互斥量来同步访问 `deque<T>` 组件。因为 `deque<T>` 组件只通过 `insert()` 和 `remove()` 成员函数修改，所以，我们可以用一个命名互斥量封装所有插入和提取。`insert()` 和 `remove()` 相应声明为：

```
template<class T> void lqueue<T>::insert(T X)
{
    lock();
    SafeQueue.push_back(X);
    unlock();
}

template <class T> T lqueue<T>::remove(void)
{
    T Temp;
    lock();
    Temp=SafeQueue.front();
    SafeQueue.pop_front();
    unlock();
    return(Temp);
}
```

如果互斥量已被占有，当调用 `lock()` 成员函数时，进程将阻塞。否则会允许线程在队列中插入数据，或从队列中提取数据。现在，插入和提取操作可以被阻塞，我们可以使用 `lqueue` 类来创建能够适当应用于多线程程序的对象。

显然，我们已经简化了这里所需要的代码来强调互斥量与它保护对象之间的关系。通过匿名互斥量，我们使用复合在 `mt_rational` 类中包含互斥量。通过命名互斥量，我们使用继承来包含互斥量与 `lqueue` 类。在两种情况下，通过在类中封装互斥量，我们可以防止大部分竞争条件和死锁。因为对象的临界区被自动锁定或被访问临界区的对象成员函数锁定，以此防止竞争条件。以两个线程更新一个队列来举例说明。一个线程在队列中插入对象。另一个线程从队列中删除对象。如果两个线程并发操作，则队列变成一个临界区。如果队列由 `lqueue` 类来表示，插入和删除操作就会受到保护。

图 8-4 中的流程图演示了两个线程，主线程和线程 A。两个线程都修改一个名叫 `TextFiles` 的 `lqueue` 对象。主线程搜索当前目录寻找包含指定模式的所有文件名。如果发现了包含指定模式的文件名，主线程就将这个文件名放进 `TextFiles`。线程 A 从 `TextFiles` 对象中删除每个文件名，并打开对应的文件。

然后搜索该文件，查找是否包含一个关键字列表。我们当然不希望在某个文件名放进 `TextFiles` 对象之前就从队列中删除了。请注意，线程 A 在继续执行之前，必须一直等到 `TextFiles` 对象中存在一个文件名。线程或进程并发执行有两种基本类型的关系。它们有 4 种同步关系以及 4 种依赖性关系。

8.1.2 同步和依赖性关系（示例）

表 8-2 列出了同步和依赖性关系以及它们是如何被控制的。

表 8-2 同步和依赖性关系以及控制机制

关 系	控 制 机 制
同步	互斥量 条件变量 条件断言
依赖性	线程间通信 进程间通信

同步关系由互斥量、条件变量以及条件断言控制。依赖性关系由线程间通信机制和进程间通信机制控制。

在图 8-4 的流程图中，线程 A 与主线程有 SS (start-to-start) 关系，因为在主线程向 `TextFiles` 中放置文件之前，线程 A 访问 `TextFiles` 对象不合适。所以，线程 A 必须等待主线程开始后才能开始。这种 SS 关系由互斥量和条件变量控制。线程 A 必须等待 `TextFiles` 对象的非空条件。一旦此条件满足，主线程将通知线程 A。不过，线程 A 在主线程释放互斥量之前不能访问 `lqueue` 对象。

线程 A 与主线程既有通信关系，也有依赖性关系。图 8-5 显示了线程 A 与主线程的这种依赖图。请注意，线程 A 不仅在通信方面，还在信息方面依赖于主线程。也就是说，在线程 A 从 TextFiles lqueue 对象删除需要的文件前，主线程必须通知 lqueue 类数据已经添加到 lqueue 对象中。竞争条件通过在主线程和线程 A 之间发生适当的锁定和取消锁定来防止。

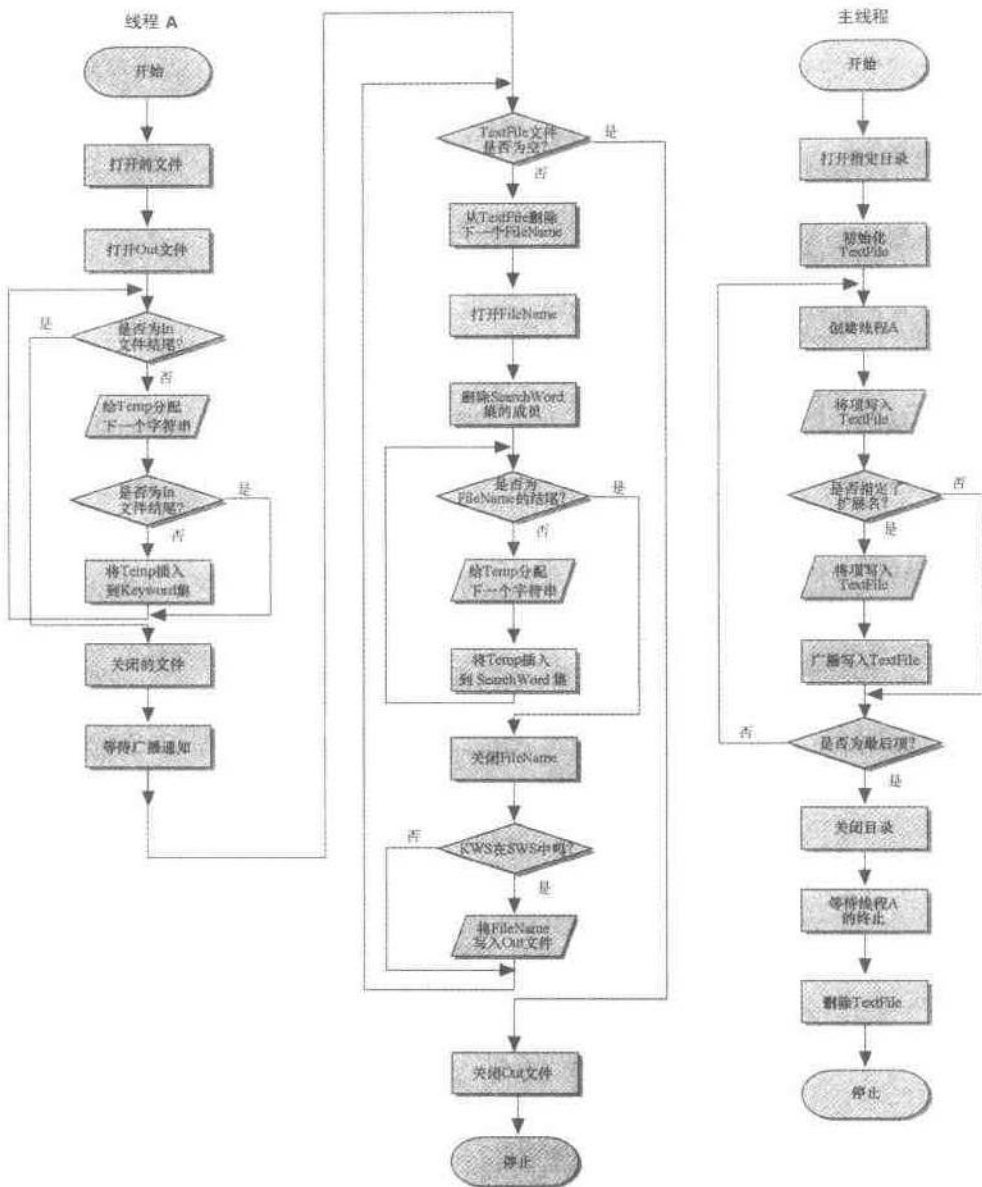


图 8-4 程序清单 8-3 中的程序的流程图

主线程与线程 A 间的关系

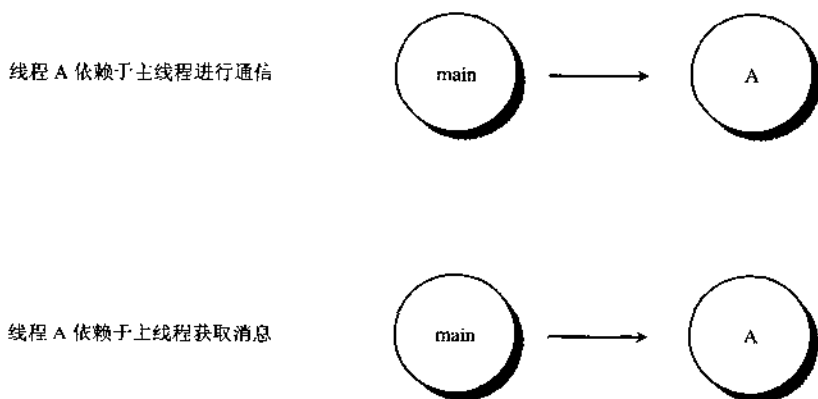


图 8-5 主线程和线程 A 之间的依赖性关系

当使用非面向对象技术的时候，这些线程间的依赖性将在每个线程内进行管理。因为两个线程都并发更新队列，程序员需要在适当的位置放入互斥量锁定以及互斥量释放调用，程序员必须按适当方式编码条件变量和断言。程序员可能在此无意间引入竞争条件和死锁。程序员可能不知道正进入临界区，所以可能不会使用互斥量。而且，程序员可能无意间使用了不平衡的互斥量调用。对互斥量锁定的每一次调用，只能有一个对对应锁定的调用，而且每一次互斥量取消锁定的调用，也必须存在一个互斥量锁定的调用。当请求锁定的数量不等于取消锁定的数量时，就在应用程序中出现不平衡互斥量调用。非平衡互斥量调用可能导致死锁。如果已经占有互斥量的线程试图再次锁定，这可能导致线程等待一个永远也不会发生的事件时阻塞自己。因为线程占有互斥量，所以线程必须首先释放互斥量。不过，如果线程已经阻塞了自己，而且等待自己释放互斥量，则出现死锁条件。图 8-5 中的线程使用面向对象技术处理共享队列临界区，然后使用过程化方法防止某些可能发生的缺陷。图 8-5 中所示两个线程代表的程序如程序清单 8-3 所示。

程序清单 8-3 代表图 8-5 流程图中两个线程的程序

```

1  #include <dirent.h>
2  #include <set.h>
3  #include <cstring.h>
4  #include <fstream.h>
5  #include <algo.h>
6  #include <string.h>
7  #include <stddef.h>
8  #include <process.h>
9  #define INCL_DOSPROCESS
10 #define INCL_DOSSEMAPHORES
11 #include <os2.h>

```



```
12 #include "ctqueue.cpp"
13
14 lqueue<string> *TextFiles;
15 set<string,less<string> > KeyWords;
16 set<string,less<string> > SearchWords;
17 void threadA(void *X);
18 string InFile;
19 string OutFile;
20
21
22 void threadA(void *X)
23 {
24     string Temp;
25     string FileName;
26     less<string> Comp;
27     ifstream In;
28     ofstream Out;
29     In.open(InFile.c_str());
30     Out.open(OutFile.c_str());
31     while(!In.eof())
32     {
33         In >> Temp;
34         if(!In.eof()){
35             KeyWords.insert(Temp);
36         }
37     }
38     In.close();
39     TextFiles->wait();
40     while(!TextFiles->empty())
41     {
42         FileName = TextFiles->remove();
43         In.open(FileName.c_str());
44         SearchWords.erase(SearchWords.begin(),SearchWords.end());
45         while(!In.eof() && In.good())
46         {
47             In >> Temp;
48             SearchWords.insert(Temp);
49         }
50         In.close();
51         if(includes(SearchWords.begin(),SearchWords.end(),
52             KeyWords.begin(),KeyWords.end(),Comp)){
53             cout << "Thread A match found in " << FileName << endl;
54             Out << FileName << endl;
55         }
56     }
57 }
```

```

58     Out.close();
59 }
60
61
62 void main(int argc, char *argv[])
63 {
64     DIR *DirP;
65     unsigned long Result = 0;
66     struct dirent *EntryP;
67     DirP = opendir(argv[1]);
68     InFile = argv[2];
69     OutFile = argv[3];
70     try{
71         TextFiles = new lqueue<string>
72             ("\\SEM32\\mymutex",0,"\\SEM32\\myevent",0,-1);
73         if(argc == 4){
74             string Temp;
75             Result = _beginthread(threadA,8192,NULL);
76             if(DirP == NULL){
77                 cerr << " Could Not Open " << argv[1] << endl;
78                 exit(1);
79             }
80             do{
81                 EntryP = readdir(DirP);
82                 if(EntryP){
83                     Temp = EntryP->d_name;
84                     if(Temp.contains(".TXT")){
85                         cout << "Found " << EntryP->d_name
86                             << " in Main Thread " << endl;
87                         TextFiles->insert(EntryP->d_name);
88                         TextFiles->broadCast();
89                     }
90                 }
91             }while(EntryP);
92             closedir(DirP);
93             DosWaitThread(&Result,DCWW_WAIT);
94             delete TextFiles;
95         }
96     }
97     catch(general_exception &X)
98     {
99         cout << X.message() << endl;
100     }
101
102 }

```

如何在多线程应用程序中使用 lqueue 对象 TextFiles 的责任从类的用户转移到了 lqueue 类的

设计者身上。

`lqueue` 类型对象的用户不必知道临界区发生的地方。用户不必担心不平衡互斥量的调用。而且，程序员不必负责完成操作系统特定 API 调用。程序员不必知道支持 `lqueue` 类型对象对应系统服务所需要的多个参数选项。这个类定义了访问临界区的方式。这是面向对象编程的一个基本概念。类通过方法（method）调控对数据组件的所有访问。在 C++ 中，使用成员函数实现方法。

在程序清单 8-3 的程序中，`lqueue` 对象的临界区在两个地方被访问。第一个访问点是 `main()` 调用以下语句时：

```
TextFiles->insert(EntryP->d_name);
```

第二个访问点是 `threadA()` 调用以下语句时：

```
FileName=TextFiles->remove();
```

两个调用都修改对象 `TextFiles`。因为两个线程并发执行，所以这构成了一个临界区，因此可能存在竞争条件。因为类 `lqueue` 的成员函数 `insert()` 和 `remove()` 自动执行锁定和取消锁定，同时并不干涉程序员使用 `lqueue` 类的对象，所以不会发生竞争条件。记住 `lqueue` 类使用继承来得到命名互斥量的功能，允许使用以下调用：

```
template<class T> void lqueue<T>::insert(T X)
{
    lock();
    SafeQueue.push_back(X);
    unlock();
}
```

```
template <class T> T lqueue<T>::remove(void)
{
    T Temp;
    lock();
    Temp=SafeQueue.front();
    SafeQueue.pop_front();

    unlock();
    return(Temp);
}
```

`lock()` 和 `unlock()` 成员函数从 `named_mutex` 类被继承。每次程序员调用 `insert()` 或 `remove()` 函数时，就会调用适当的 `lock()` 和 `unlock()` 对。这消除了程序员在使用 `lqueue` 类时对竞争条件可能发生的担心。虽然 `lqueue` 类防止修改双端队列组件的操作同时发生，但 `lqueue` 类的用户仍然可能得到预想不到的结果。如果一个线程为 `inserting()` 或 `removing()` 元素，与此同时，另一个线程正锁定在队列的前端，程序员可能不会得到预期的结果。这个问题的简单解决方案是对所有读取 `dequeue` 组件的成员函数提供锁定和取消锁定处理，对修改 `deque` 组件的成员函数也是如此。不过，这样做的开销较大。线程安全程序要求在安全和效率之间寻求适当的平衡点。对于程序清单 8-3 中的程序，当两个线程都访问 `cout` 对象时，它还面对另外一个潜在的竞争条件。在这个程序中，`cout` 不受互斥量保护，而且两个线程都允许同时修改 `cout`。这可能导致错乱的输出显示。

我们讨论了这个程序中两个线程间的 SS 关系。不过，在这两个线程间还存在一种 FF

(finish-to-finish) 关系。lqueue 对象 TextFiles 由程序中的第一个线程自动创建。然后，TextFiles 对象由程序中的第一个线程删除。如果第一个线程在第二个线程完成之前完成并删除 TextFiles 对象，就发生内存违规、页错误或其它同等的致命错误。我们不能允许第一个线程在第二个线程完成前才完成。因此我们说主线程与线程 A 之间存在 FF 关系。直到线程 A 完成后，主线程才能完成。所以，虽然主线程没有任何处理过程要完成，但它必须一直等到线程 A 的完成，然后主线程在不创建内存违规条件的前提下删除 TextFiles 对象。注意在这个小型程序中，使用多个线程时存在多个潜在的缺陷。图 8-6 显示了这个例程中涉及的组件条块图，同时显示哪一个线程可以访问这些组件。

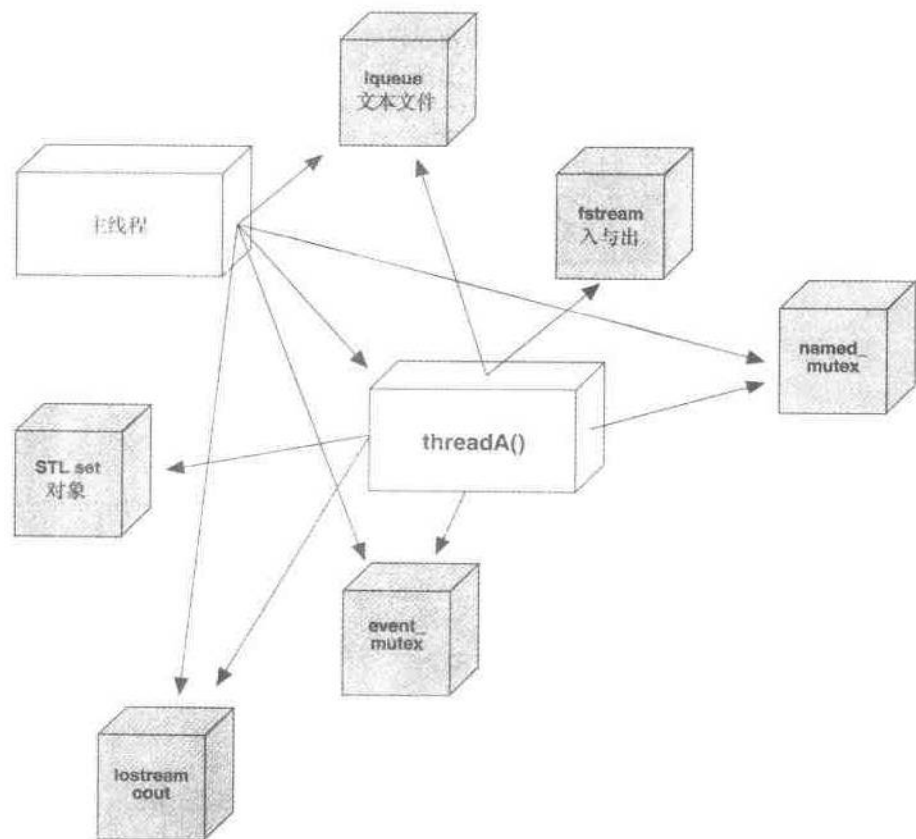


图 8-6 显示程序清单 8-3 中程序的组件以及组件间通信的条块图。这些组件有：主线程、threadA、STL 对象、named_mutex 类、event_class 类、lqueue 类、iostream cout 以及 fstream 输入和输出对象

每个组件必须服务于程序的正确多线程化。在本书的余下章节中，在使用以上组件及其它类似组件时，我们将重新定义多线程处理技术。请记住我们的目标是通过面向对象技术管理竞争条件和死锁。同样，我们希望将实现并发的责任转移到类、类库以及应用框架级别，减少并发和同步编程的一些复杂性。

8.1.3 表示条件的类

在程序清单 8-3 的程序中, `TextFiles` 对象有一个 `wait()` 成员函数, 它导致调用线程一直阻塞到某事件的发生。从这种意义上说, `wait()` 成员函数充当一个互斥量。通过互斥量, 调用线程如果试图锁定已经被占有的互斥量, 它就会阻塞。这里的阻塞由一个条件产生, 这个条件的值为 `True` 或 `False`。程序清单 8-3 程序中的 `Iqueue` 类继承了两个私有类。第一个类是 `named_mutex` 类。第二个类是 `event_class`。 `Iqueue` 类与 `event_class` 之间的类关系图如图 8-7 所示。 `event_class` 是另一个接口类。它封装了一个事件互斥量。调控互斥量代表占有权, 事件互斥量代表条件。

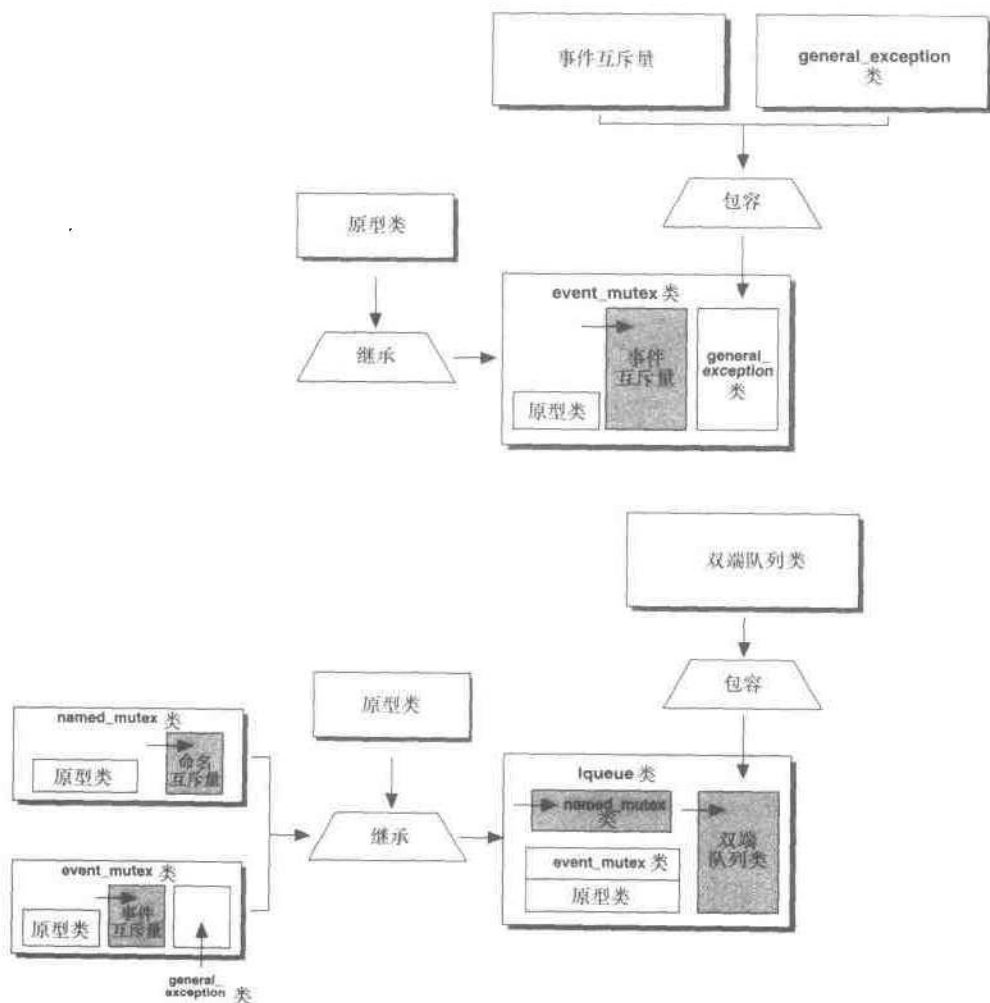


图 8-7 lqueue 类与 event_class 类的类关系图

事件互斥量有两种状态。该状态通常被冠以不同的名字：set、unset、true、false、posted、not posted 以及其它一些名字，用于描述一个双重状态变量（two-state variable）。

当一个调用线程想等待某互斥量变量，而互斥量变量为 unset 时，线程将被阻塞，一直到另一个线程设置了此互斥量为止。相反，如果调用线程想等待互斥量变量，且事件互斥量为 set，它不会阻塞。POSIX 线程、Win32 以及 OS/2 对待事件互斥量稍有不同。尽管在所有 3 种环境中，事件互斥量的主要功能都是通过信号机制同步，但实现同步的方法有细微的差别。表 8-3 列出了 POSIX、Win32 以及 OS/2 环境中的事件互斥量系统 API（这里的事件互斥量即可以在任一种环境下实现）。

表 8-3 POSIX、Win32 以及 OS/2 环境中的事件互斥量系统 API

环 境	对事件互斥量的操作	描 述
POSIX	pthread_cond_init()	初始化一个条件变量
	pthread_cond_destroy()	销毁一个条件变量
	pthread_cond_wait()	挂起调用线程，一直到另一个线程通知条件变量
	pthread_cond_timedwait()	挂起调用线程，一直到另一个线程在指定时间内通知条件变量。如果在指定时间内没有通知条件变量，则从等待中释放线程
	pthread_cond_signal()	为等待锁定某互斥量的线程创建一个规划次序。将锁定的互斥量赋予线程。将其它所有线程放到等待获取该互斥量的队列中
	pthread_cond_broadcast()	释放所有等待条件变量的线程。实际上只有一个线程得到互斥量的占有权
OS/2	DosCreateEventSem()	创建一个事件信号量
	DosOpenEventSem()	打开一个事件信号量
	DosPostEventSem()	发送一个事件信号量
	DosResetEventSem()	重置一个事件信号量
	DosWaitEventSem()	等待事件信号量的发送
	DosCloseEventSem()	关闭一个事件信号量
	DosQueryEventSem()	返回事件信号量的发送计数。发送计数就是被发送信号量的次数
Win32	CreateEvent()	创建一个事件信号量对象。如果将某个名字指定为它的参数，则创建一个命名事件信号量对象
	OpenEvent()	将对象的名字用作它的参数之一，从一个无关进程打开一个事件信号量对象
	SetEvent()	将事件信号量对象的状态更改为被通知。如果事件已经被通知，函数将无作用
	ResetEvent()	重置事件信号量对象。它更改对象的状态为被通知

续表

环 境	对事件互斥量的操作	描 述
	PulseEvent()	将对象的状态更改为被通知。释放所有等待线程，然后将对象的状态更改为未被通知
	WaitForSingleObject()	导致线程一直等待到对象被通知。将对象指定为参数之一
	WaitFormultipleObjects()	导致线程等待一个包含一个或多个同步对象的数组

三种环境的重要不同点与多线程等待条件时的发生情况有关。在 POSIX 环境中，基本事件互斥量或条件变量通常只取消锁定一个线程。这常常是被阻塞的第一个线程。而在 OS/2 环境中，则取消等待条件的每个线程。对于涉及被互斥量阻塞的临界区的场合，只是第一个接触互斥量的线程获得它的占有权，其它线程被阻塞。如果在事件变量创建过程中请求人工重新设置，Win32 环境可能取消阻塞所有的线程。否则，它只取消阻塞一个线程，而且事件变量立即被重新设置，导致其它线程保持阻塞。

每种环境基于事件互斥量在该环境中的常用方式来作出设计决策。不过，事件和条件变量的使用方式有多种。不存在对于如何使用的设置规则。事件类的设计者决定封装哪一个模型。在任何一种环境的条件变量和断言实现中使用的基本元素可用于模拟其它两种环境，所以，程序员自由选择所需要的方式。在这里，我们只对条件的基本概念建模，它要么是 set，要么是 not set。如果条件是 set，调用线程将阻塞到另一个线程调用提示、通知或设置此条件。否则，将允许线程继续处理过程。下面是 event_class 的类声明：

```
#include "eclasses.h"
#include <windows.h>

class event_mutex{
protected:
    LPSECURITY_ATTRIBUTES SecurityPtr;
    HANDLE EventMutex;
    char MutexName[81];
    int InitallySet;
    general_exception EventException;
    unsigned long EventWait;
    unsigned int AccessLevel;
    int ManualReset;
public:
    event_mutex(char *MName,unsigned int Initial=0,
        unsigned long Reset,LPSECURITY_ATTRIBUTES Secure);
    event_mutex(char *MName,unsigned int Access,
        unsigned long Inherit);
    void postEvent(void);
    void waitEvent(void);
    void resetEvent(void);
```

```

    ~event_mutex(void);
};

```

针对以上声明, 我们使用 Win32 系统服务。虽然对于 POSIX 和 OS/2 环境的函数调用不同, 但使用 event_mutex 类时, 成员函数意义的使用方式一致。程序清单 8-4 包含 event_mutex 类的 Win32 定义。

程序清单 8-4 event_mutex 类的定义

```

#include "ctevent.h"

event_mutex::event_mutex(char *MName, unsigned int Initial = 0,
                          long Reset, LPSECURITY_ATTRIBUTES Secure);
{
    EventMutex = CreateEvent(Secure, Reset, Initial, MName);
    if(!EventMutex){
        EventException.message("Could Not Create Event Semaphore");
        throw EventException;
    }
    if(MName != NULL){
        strcpy(MutexName, MName);
    }
    EventWait = INFINITE;
}

event_mutex::event_mutex(char *MName, int Inherit, long Access)
{
    EventMutex = OpenEvent(Access, Inherit, MName);
    if(!EventMutex){
        EventException.message("Could Not Create Event Semaphore");
        throw EventException;
    }
    if(MName != NULL){
        strcpy(MutexName, MName);
    }
    EventWait = INFINITE;
}

void event_mutex::postEvent(void)
{
    SetEvent(EventMutex);
}

```



```
    }

    void event_mutex::waitEvent(void)
    {
        WaitForSingleObject(EventMutex,EventWait);
    }

    void event_mutex::resetEvent(void)
    {
        ResetEvent(EventMutex);
    }

    event_mutex::~event_mutex(void)
    {
        if(!CloseHandle(EventMutex)){
            EventException.message("Could Not Close Mutex");
            throw EventException;
        }
    }
```

请注意，程序清单 8-4 中的成员函数只是 Win32 系统服务的封装器。封装技术的目的是封装事件系统服务（event system service），简化事件同步的使用，而且防止一定类型的死锁。同时，通过使用接口类技术来封装锁定和同步基本元素，程序员可以开发一种面向对象的系统独立途径来多线程化应用程序。因为通过继承，lqueue 类既有一个 event_mutex，还有一个 named_mutex，所以 deque 组件可以通过条件和锁定来得到保护。lqueue 成员函数调用它的两个互斥量类的成员函数。这些成员函数利用了 named_mutex 和 event_mutex 类。

```
template <class T> void lqueue <T>::insert(T X)
{
    lock();
    SafeQueue.push_back(X);
    unlock();
}

template <class T> T lqueue <T>::remove(void)
{
    T Temp;
    lock();
    Temp=SafeQueue.front();
    SafeQueue.pop_front();
    unlock();
}
```

```

        return(Temp);
    }

    template <class T> void lqueue <T>::wait(void)
    {
        waitEvent();
    }

    template <class T> void lqueue <T>::broadcast(void)
    {
        postEvent();
    }

```

这4个成员函数允许在多线程环境中使用 `lqueue` 类型对象。`lqueue` 对象的构造函数接受事件互斥量和命名互斥量的名字。通过使用这些名字，其它进程中的线程可以访问这些互斥量。在程序清单 8-3 中的程序中，事件互斥量用于通知 `threadA()`，`TextFiles` 对象中至少有一个文件名。实现的代码段如下所示：

```

if(Temp.contains(".TXT")){
    cout<<"Found"<<EntryP->d_name<<"in Main Thread"<<endl;
    TextFiles->insert(EntryP->d_name);
    TextFiles->broadcast();
}

```

一旦 `TextFiles` 拥有了一个文件名，它就会广播它不再为空的消息。此时，在如下代码处阻塞的 `threadA()`：

```
TextFiles->wait();
```

将取消阻塞，并继续其处理。虽然这个 `wait()` 成员函数阻止 `threadA()` 最先从空队列中删除一个文件名，但它并不阻止 `threadA()` 在以后从空队列中删除文件名。例如，如果主线程插入了一个文件名，然后通知 `threadA()` 队列已经有了一个文件名，于是 `threadA()` 删除这个文件名，而且在主线程插入另一个文件名前试图删除另一个文件名，`threadA()` 处理过程终止。事实上，`threadA()` 可能比主线程执行快，这意味着虽然主线程可能有多个文件要处理，但如果在 `threadA()` 的处理过程中，`TextFiles` 变成了空，它就会退出如下循环：

```

while(!TextFiles->empty())
{
    File name= TextFiles->remove();
    in.open(File name.c_str());
    SearchWords.erase(SearchWords.begin(),SearchWords.end());
    while(!In.eof() && In.goog())
    {
        In>>Temp;
        SearchWords.insert(Temp);
    }

    In.Closed();
}

```

```

        if (includes(SearchWords.begin(), SearchWords.end(),
                     KeyWords.begin(), KeyWords.end(), Comp)) {
            cout << " thread A match found in " << File name << endl;
            Out << File name << endl;
        }
    }
}

```

为了让处理过程正确进行，`threadA()`必须等待主线程找到它要处理的所有文件，或者等待主线程每次发现一个文件后的通知。这可以让 `threadA()`在删除文件名前调用 `reset()`，在删除另一个文件名前调用 `wait()`来完成。

`named_mutex` 和 `event_mutex` 类的构造函数是在构建封装系统服务的接口类时可以使用的关键技术。许多 IPC 或锁定系统服务允许其它进程访问互斥量、条件变量、管道或共享内存。创建初始组件的线程或进程通常设置该组件的特征。例如，创建一个命名管道的进程或线程称做服务器，它负责设置该管道的属性。客户进程不需要创建使用的管道。客户只需要知道管道的名字，然后客户就可以打开管道。因为接口类在构造函数中封装了管道创建，管道可以在创建对象时得以创建。对于服务器，这种工作方式没有问题，但我们不希望客户使用同样的构造函数，所以，我们声明被客户调用的第二个构造函数。这允许我们让服务器和客户使用同一个接口类。`event_mutex` 类有两个构造函数：

```

event_mutex:: event_mutex(char *MName, int Inherit, long Access)
{
    EventMutex=OpenEvent(Access.Inherit, MName);
    if(!EventMutex){
        EventException.message("Could Not Create Event
                               Semaphore");
        throw EventException;
    }

    if(MName!=NULL){
        strcpy(MutexName, MName);
    }
    EventWait=INFINITE;
}

event_mutex:: event_mutex(char *MName,unsigned int Initial=0,
                          long Reset, LSECURITY_ATTRIBUTES Secure);
{
    EventMutex=CreateEvent(Secure, Reset, Initial, MName);
    if(!EventMutex){
        EventException.message("Could Not Create Event
                               Semaphore");
        throw EventException;
    }
}

```

```

    if (MName != NULL) {
        strcpy(MutexName, MName);
    }
    EventWait = INFINITE;
}

```

第一个构造函数由创建初始事件互斥量或条件变量的线程使用。调用这个构造函数时，当前进程中的任何线程都可以利用这个事件互斥量。如果 `MName` 参数包含一个有效名字，那么事件互斥量可以被其它进程中的线程使用。不过，当其它进程中的线程想共享这个事件互斥量时，其它进程不会使用第一个构造函数创建事件对象。相反，此时使用第二个构造函数。注意，第二个构造函数调用与 `CreateEvent()` 相反的 `OpenEvent()`。这允许其它进程访问现有事件互斥量。打开事件互斥量只是第一步，进程必须接着调用 `wait()`、`broadCast()` 或 `reset()` 成员函数来实际应用事件互斥量。使用多构造函数的模式使得客户/服务器多进程处理更方便。它避免程序员必须编写重复代码之劳。

8.1.4 等待多个事件或互斥量

另一种类型的条件变量就是包含多个事件或互斥量的事件互斥量 (event mutex)。阻塞于这些多个等待变量的线程可能等待所有事件，在处理过程继续之前，互斥量根据这些事件来设置。例如，`lqueue` 对象可能用于保存表示不同类型文件的文件名。这些文件包括即将处理的文本文件、程序文件以及多媒体文件。我们希望处理多媒体文件的方式与处理文本文件和程序文件的方式不同。我们可以使用多个条件互斥量来协助导向处理过程。如果线程 1 在队列中插入文件名，线程 2 从队列中删除文件名。我们必须确保线程 1 在队列中放置文件名之前，线程 2 不会删除文件名。这需要线程 1 在队列包含文件名时，广播通知线程 2。将这个条件命名为 `NotEmpty`。线程 1 也负责决定与每个文件名关联的是哪种文件类型。为了让线程 2 知道什么时候处理多媒体文件，在线程 1 在队列中插入一个表示多媒体文件的文件名时，它必须广播此消息。将此条件命名为 `Multimedia`。所以，线程 2 等待包含两个条件的事件：

```
wait(NotEmpty, Multimedia)
```

线程 2 将阻塞于 `wait()`，直到线程 1 广播通知了这两个条件。这是一个非常重要的功能。它可以用于防止导致死锁的情形。一个典型的死锁例子就是，有两个并发执行的线程，名叫线程 A 和线程 B，当它们竞争同一个资源时发生的情形。两个线程想同时获取资源。不过，它们按不同的顺序获取资源，或者一个线程执行得比另一个线程快，它可能首先得到资源之一。例如，两个线程可能都需要访问分配给本地声卡的端口以及分配给本地 NTSC 卡的端口。NTSC 卡用于记录和显示视频。声卡负责显示和记录 `.mod`、`.wav` 和 `.midi` 文件，NTSC 卡负责记录和显示完整动画视频。线程 A 可能锁定 NTSC 卡，而线程 B 锁定声卡。线程 A 在等待线程 B 释放声卡时，它可能占有 NTSC 卡。线程 B 可能在等待线程 A 释放 NTSC 卡时占据声卡。每个线程在得到所需资源并完成执行前，它不会释放占有的资源。两个线程都等待着一个可能永远也不会发生的事件。这种情况描述了一种典型的死锁现象。多个条件或多个事件互斥量可以用于防止这类死锁。

如果线程 A 和线程 B 阻塞于多个事件互斥量，则两个线程都不能获得资源锁定，除非事件列表中的所有资源都可用。对于多个事件互斥量，这是一种要么都能获得资源锁定，要么都不能获

得资源锁定的情形。在线程可以继续前，所有的条件必须都被设置。可用多个条件变量使线程等待多个事件或多个信号量。下面是一个多个事件类的声明：

```
class multiple_event_mutex{
protected:
    char MutexName[81];
    SEMRECORD *ConditionList;
    int EventNumber;
    long int WaitSelection;
    HEV EventMutex;
    MultipleEventMutex;
    general_exception EventException;
    int Duration;
public:
    multiple_event_mutex(char * MName,int ENum,
        long int WaitType,int Initial);
    multiple_event_mutex(char * MName);
    ~ multiple_event_mutex(void);
    int duration(void);
    void duration(int X);
    void postEvent(int X);
    void waitEvents(void);
};
```

这是一个封装了在 OS/2 中可用的多个事件互斥量服务的接口类。与 `mutex`、`named_mutex` 和 `event_mutex` 类一样，这个类要用作一个数据类或设备类的组件。例如，我们可以使用这个类作为 `lqueue` 类的一部分。`lqueue` 类的用户因此可以将多个事件与 `lqueue` 类型的对象使用关联起来。程序清单 8-5 包含 `multiple_event_mutex` 类的定义。

程序清单 8-5 `multiple_event_mutex` 类的定义

```
multiple_event_mutex::multiple_event_mutex(char *MName,int ENum,
        Long int WaitType,
        Int Initial)
{
    int N;
    EventNumber=EEnum;
    ConditionList=new SEMRECORD[EEnum];
    For(N=0;N<EEnum;N++)
    {
        if(DosCreateEventSem(NULL,&EventMutex,0,Initial)){
            EventException.message("Could Not Create Event
                Semaphore");
            Throw EventException;
        }
        ConditionList[N].hsemCur=&EventMutex;
        ConditionList[N].ulUser=N;
    }
}
```

```

    }
    if (DosCreateMuxWaitSem(MName, &MultipleEventMutex,
                           sizeof(ConditionList),
                           ConditionList, WaitType)) {
        EventException.message("Could Not Create Multiple
                               Event Semaphore");
        throw EventException;
    }
}

multiple_event_mutex::multiple_event_mutex(char *MName)
{
    if (DosOpenMuxWaitSem(MName, &MultipleEventMutex)) {
        EventException.message("Could Not Open Event Semaphore");
        throw EventException;
    }
}

multiple_event_mutex::~multiple_event_mutex(void)
{
    if (DosCloseMuxWaitSem(MultipleEventMutex)) {
        EventException.message("Could Not Close Event Semaphore");
        throw EventException;
    }
    delete ConditionList;
}

int multiple_event_mutex::duration(void)
{
    return(Duration);
}

void multiple_event_mutex::duration(int X)
{
    Duration=X;
}

void multiple_event_mutex::postEvent(int X)
{
    EventMutex=(HEV)ConditionList[X].hsemCur;
    DosPostEventSem(EventMutex);
}

void multiple_event_mutex::waitEvents(void)
{
    unsigned long User;

```

```
DosWaitMuxWaitSem(MultipleEventMutex, SEM_INDEFINITE_WAIT,  
                  &User);  
}
```

Win32 环境通过使用 `WaitMultipleObjects()` 系统服务也支持这种类型的类。

对 ACID 使用多个事件互斥量

要求永久对象的面向对象数据库和应用程序中存在许多多线程处理和多个事件互斥量的可能性。这些环境通常需要按永久格式保存的对象间的复杂动态关系。这种永久性可能存在于外部存储器，也可能存在于虚拟存储器中。这些永久对象可能在网络化环境中被访问。面向对象数据库通常用于支持客户/服务器处理模式。因为每个对象可能提供多种服务，所以对象同时被多个线程使用是不可避免的，每个线程可能使用着不同的对象服务。这种多重访问可能创建竞争条件。在面向对象数据库级别创建的竞争条件与传统竞争条件相似，它们都可能让数据处于一种不可靠状态。不过，在面向对象数据库环境中创建的竞争条件通常是由同时复杂对象更新产生的。我们至今所谈到的竞争条件通常涉及单个内存位置或单个内存块。通过永久对象创建的竞争条件通常涉及多个数据库，这些数据库需要基于动态，有时基于一套复杂的条件来更新。更新影响的每个数据库或对象通常直接或间接关联。

例如，有一个数据库跟踪进入某医院急救室的患者。每个进入的病危者都会影响多种资源。急救室面向对象数据库包含为急救室、床位、外科医生、护士、技师、医疗设备、药物、血液供应、病人、抢救医疗过程以及付费信息建模的对象。这个面向对象数据库用于评价医院在某时刻抢救病人的能力。所以，面向对象数据库用作一个实时决策工具。基于面向对象的状态，决定让抢救室接收另外的急救病人，或不接收另外的急救病人。每个接收的病人在这些对象中创建了一个事务。一旦接收了急救病人，急救室的床位数量就会减小、在位外科医生的工作负荷就会减少。必须分配医疗设备。必须准备药物。可用血液被消耗，病人立即开始支付医疗费用。随接收病人的增加，急救室要同时补充血液供应，请求更多的外科医生、购买新医疗设备等等。因为每个事务影响了急救室模型的如此多个方面，而且依赖于可用资源，接收病人与医疗资源补给之间必须保持协调。虽然在某一时刻可以多接收一个病人，但不能让竞争条件或死锁导致急救室模型变得不可靠。图 8-8 是面向对象急救室及其所有组件间依赖性的条块图。

每种资源——床位、外科医生、医疗设备、血液供应、护士、技师以及药物——都必须在每个事务中进行更新。一旦事务开始，我们不希望它被打断。数据库处理必须更新所有的资源，或不应当更新任何资源。这些复杂的更新称做事务 (transaction)。事务中不能容忍竞争条件和死锁。

针对这种复杂事务处理，有一套标准可以衡量特定事务是否让数据库处于一致的状态。其中一个标准就是 ACID 测试。缩写 ACID 代表原子性 (atomicity)、一致性 (consistency)、孤立性 (isolation) 以及持久性 (durability)。

通过了 ACID 测试的事务在多线程、并行处理、或多用户环境中就是安全的。如果事务具有原子性，它就会完整执行，或者根本不执行。一致性处理事务中的数据完整性。如果事物保证所有的数据断言、前置条件以及后置条件在更新中能得到维护，就认为它具有一致性。孤立性序列化或模块化更新。实质上，孤立通过确保没有数据丢失、对数据元素一次只进行一个修改来防止竞争条件。持久性通常是一种恢复方案。在硬件或软件崩溃时，持久性保证数据库可以恢复到一

致性的状态。

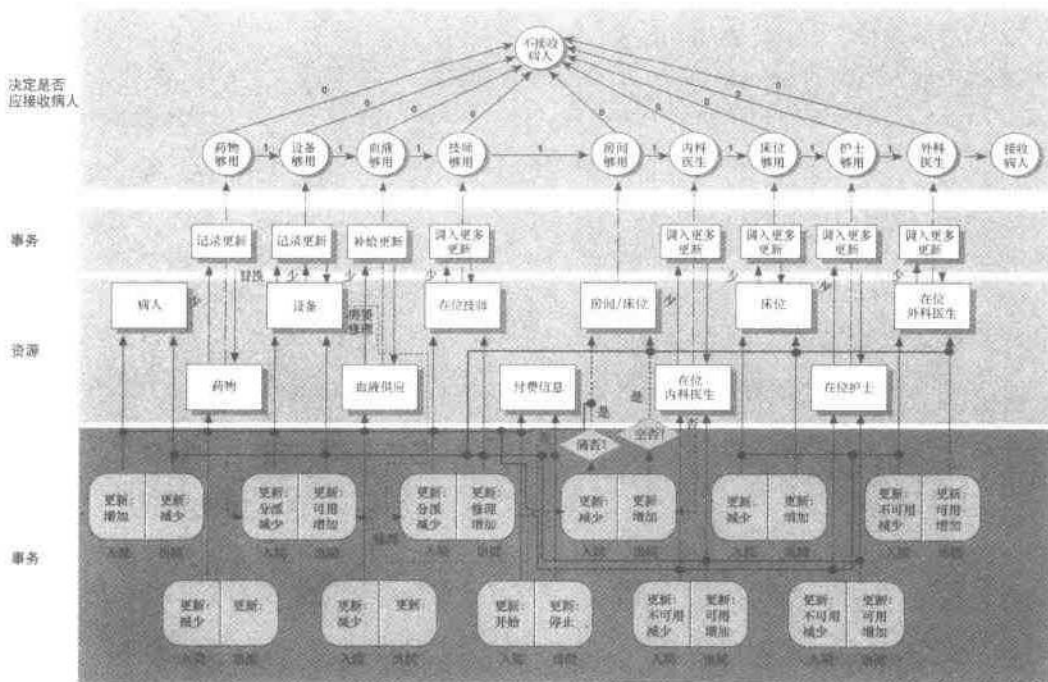


图 8-8 面向对象急救室及其所有组件间依赖性的条块图

多个事件互斥量可以帮助程序员实现 ACID。因为多个事件互斥量可能导致某线程阻塞到所有资源列表中的资源都可用为止，所以，多个事件互斥量可用于保护关键性任务的更新。例如，我们可以用一个互斥量封装急救室的每种资源。然后为病人接收线程编写代码，使用多个事件互斥量让线程阻塞，直到每种资源都被锁定为止。

8.1.5 通过类成员函数锁定和取消锁定

到目前为止，我们已经介绍了 4 个同步组件：

- mutex 类；
- named_mutex 类；
- event_mutex 类；
- multiple_event_mutex 类。

一般而言，这些类型的类倾向于用作其它类的组件。例如，我们使用继承在 lqueue 类中包含 named_mutex 和 event_mutex。然后，我们在 lqueue 类的 insert() 和 remove() 成员函数中使用 mutex 类的 locking() 和 unlock() 成员函数。也就是说，我们显式调用 lock()、unlock()、post() 或 wait() 函数。显式使用 locking() 函数是类中嵌入互斥量的用法之一。类中嵌入互斥量的另一个用途自动完成。

通过互斥使用构造函数和析构函数

lock()、unlock()、post()和 wait()类型成员函数可以放置在它们宿主类的构造函数和析构函数中。因为构造函数和析构函数被自动调用，所以无需程序员的进一步干预就可以发生互斥处理。在析构函数中放置适当的 unlock()或 post()类型成员函数后，只要删除宿主对象，或者越出作用域，unlock()或 post()类型函数将被自动调用。在构造函数和析构函数中放置互斥量函数是另一种防互斥量不平衡调用的途径。前面讲过，不平衡调用互斥量可能导致竞争条件和死锁。我们可以声明某个类如下：

```
class A: private mutex{
public:
    A(void){lock();}
    ~A(void){unlock();}
};
```

每次声明一个 A 类型的对象，它将锁定互斥量并允许对象的构造函数完成，或对象的析构函数阻塞到释放互斥量为止。有两个地方对这类锁定和取消锁定特别方便。第一个地方是在 C++ 块中。C++ 语言允许变量或对象在任何复合语句或块中声明。所以，我们可以编写如下函数：

```
void myFunc(void)
{
    int Count;
    for (Count=0; Count<N; Count++)
    {
        A Value;
        Value++;
    }
}
```

当 Value 在 for 循环中创建时，Value 的互斥量自动被锁定。如果 Value 不能锁定它的互斥量，它将阻塞到能锁定为止。for 循环完成后，互斥量通过 Value 的析构函数自动取消锁定。所以互斥量锁定和取消锁定对 Value 的用户都是透明的。

可以利用构造函数和析构函数自动锁定和取消锁定互斥量的第二个地方位于 try 块内。请看如下代码：

```
void myFunc(void)
{
    try{
        A Count;

        //其它执行代码
        Count++;
    }
    catch(...) {
    }
}
```

以上代码声明 `Count` 为 `A` 类型的对象。声明 `Count` 后，互斥量被锁定。如果由于某些原因 `myFunc()` 的处理过程中抛出了异常，`Count` 的析构函数将自动调用，自动释放被 `Count` 锁定的互斥量。所以，通过锁定、取消锁定、等待或通知类型函数，可以使用常规成员函数，也可以使用构造函数和析构函数。如果需要锁定与取消锁定嵌入互斥量或条件变量，或在 `C++` 块内多次设置（`set`）与重新设置（`reset`），则在调用的常规函数内使用互斥量操作。不过，如果互斥量操作只需要一次性激活或禁止，在这种情况下可以考虑使用构造函数和析构函数。

8.1.6 小结

同步对象可以通过设计封装了系统 API 的接口类（`interface class`）来创建。一旦创建了同步类（`synchronization class`），在那些被多线程或多进程并发或同步使用的任何类型 `C++` 类中，可以添加以上同步类。在某个类中包含同步类或者使用复合、包容或继承来实现。建议只让类中的直接成员访问同步类。只有在非常特定的条件下才允许同步类被派生类访问。一般而言，嵌入同步类的成员函数和数据成员不应该为公有。通过封装同步对象，避免对同步对象的无意操作。在类成员函数内部执行锁定和取消锁定，类临界区自动受保护。类的概念要求由类的设计者负责在多线程环境中访问类的数据。可用多个类构造函数将宿主类（`host class`）配置成客户或服务器。通过将 `locking()` 和 `unlocking()` 放入构造函数和析构函数，可以让同步对象更安全，也更容易使用。在 `try(){}` 块中放入声明对象的优点在于，抛出异常时自动调用它们的析构函数。应该用面向对象技术来关联同步机制与即将同步的对象或设备。也就是说，在任何可能的地方，避免自由漂浮的同步对象。使用类聚集（`class aggregation`）来连接同步类（`synchronization class`）与数据（`data class`）和设备类（`device class`）。

线程处理面向对象架构

似乎要发生的是，实体像“幽灵”一样相互间自由传递，并在一系列操作的基础上交互，这些操作与真实世界存在不真切的对应。随着我们对这个新生态系统的熟悉，我们对现实物理的原始渴求在很大程度上已经消失。

Liquid Architectures in Cyberspace

——Marco Novak

软件片断的架构（architecture）就是一套控制软件操作的规则、模式、进程、执行协议以及断言。软件架构表示整个软件的结构，与数据组织与执行流程相关。架构反映设计思想、开发方法学以及域模型。到目前为止，我们已经涉及了用于构建多线程架构的 C++ 组件。到最后，我们将看到 C++ 组件是如何用于构建面向对象多线程架构的。不过，我们首先要定义多线程软件架构（multithreaded software architecture）。

9.1 什么是多线程架构

多线程软件架构是一种将工作模式分解为两个或更多并发执行线程的软件架构。分解软件（factoring software）是分割为单独逻辑任务的过程，供软件的工作模式来执行。其中部分任务分配给不同的执行线程。当这些线程允许同步或并发在单进程内执行时，软件就具有多线程架构。这在很大程度上是由于软件的数据组织和执行流程依赖于对应的并发或并行化模型的可用性。

程序中的并行与多线程架构

我们在此要区别对待包含并行算法与多线程化进程的计算机程序。这种区别只是一种程度上的区别，而不是一种技术的区别。在解决并行算法所解决的问题时，它包含能够并发执行的指令。多线程将一个进程分解成逻辑数量的并发执行任务，而并行算法将一个任务分解成并发执行的指

令。有时,这些指令将进一步分解成并发执行的子指令(subinstruction)。尽管计算机程序(computer program)、算法(algorithm)以及进程(process)这些概念的界限通常模糊不清,但为了理解什么是多线程架构,将它们区分开来很重要。图 9-1 简单描述了计算机程序、算法以及进程间的关系。算法内的基本执行单元是一条语句(有时是一条子语句),进程内的执行基本单元是线程(thread)。

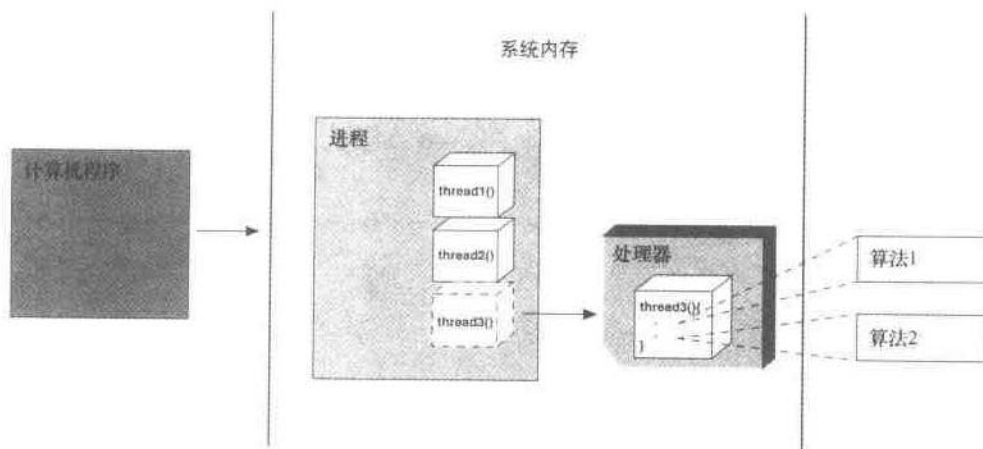


图 9-1 简单描述计算机程序、算法以及进程间的关系

为了帮助区分多线程架构与其它类型的并发模型,我们在若干常用执行单元间划上了明确的界限。例如,为了便于讨论,我们指定对话(session)为最大的执行单元。对话包含一个或多个可以并发执行的进程。当有多个进程并发执行时,对话就是多任务或多进程。一个进程包含一个或多个线程。当一个进程内有两个或更多线程并发执行时,进程就是多线程(或称多线程化)。一个线程包含一条或多个指令。当一个线程内包含一条或多条并发执行的指令时,这些指令共同形成了一个算法,我们称这套指令为并行算法(parallel algorithm)。图 9-2 显示了软件系统内基本执行单元的关系。

当然,图 9-2 对这些执行单元的划分过于简单化了。有时,对于那些没有分别的地方,它可能作了区分。有时在进程和对话或者线程和进程间没有明显的区别。而且,这里隐式假定所有的线程都至少包含一个算法,在某些情况下,可能要求多个线程完成整个系统包含的唯一算法。在实际应用中,这些区分通常模糊不清。不过,我们希望能将多线程架构指定为一种处理较大实体的并发形式,它所处理的实体比并行算法处理的实体要大。多线程架构被分解成线程,并行算法被分解成单条指令。计算机程序可能包含没有被多线程化的并行算法,进程没有并行算法地多线程化。对于并行算法是否真正由多线程实现,这方面的讨论很热烈。当讨论并行算法时,所讨论的单元指单条指令或子指令。当讨论多线程(或多线程处理)时,所讨论的单元通常指包含一套指令的逻辑任务,这套指令以过程或函数的形式出现。语言结构通常都支持并行算法。操作系统服务或线程库通常支持多线程架构。并行算法通常面对并发的微观方面,而多线程面对并发的宏观方面。这种区别是一种认识程度上的区别,而不是技术上的区别。

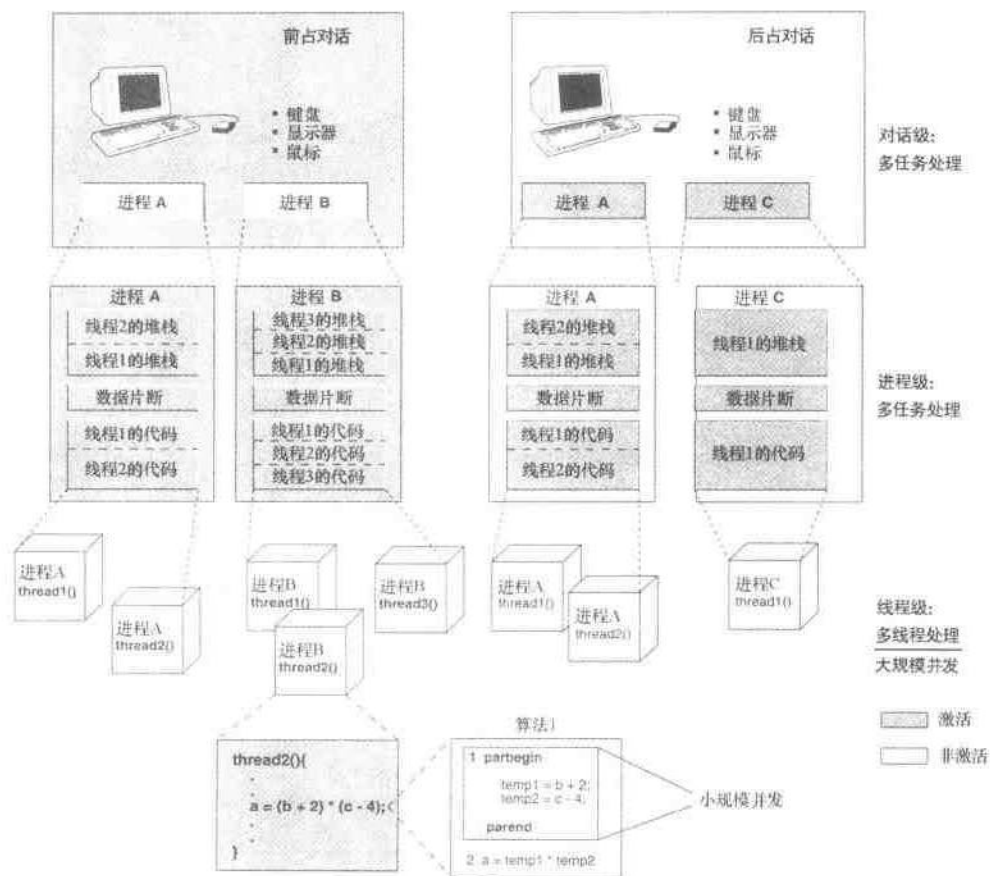


图 9-2 执行单元的过于简单化。在有时不存在分别的地方进行了区分

9.2 使用多线程的常见架构

一旦选择了软件片断的架构，就可以根据这一架构的假设和协议来进一步推进设计和实现决策。有些架构天生就适于多线程处理。当程序员在一个系统或应用程序中寻求使用多线程处理时，多线程处理必须遵循此架构。也就是说，在一个中到大型的系统中，一个成功的多线程处理应用程序不能本末倒置。按一种特别方式来使用多线程处理将导致不能维护、改进或理解的脆弱系统。在应用程序中使用多线程处理的目标之一是精选出具有内在并发性和并行性的架构。通过这种方式，软件开发不必强迫应用程序的逻辑使用多线程化模型。至于多线程处理，功能必须服从形式。应当搜寻和使用那些天生就适合于多线程处理的架构。如果对于研究域不存在这样的架构，则必须创建这样的架构。多线程处理应当服从该架构；架构不应受多线程处理的驱动。有一些天生就适合于多线程处理的常见架构。下面是其中的三种架构：

- 客户机/服务器;
- 事件驱动编程(GUI);
- 黑板 (blackboard)。

客户机 (client) 和服务器 (server) 的术语应以多种方式应用于许多不同类型的软件应用中, 因为客户机/服务器范例是一种强大和灵活的组织形式。客户机/服务器范例将工作模式分为由进程或线程表示的两端。一端 (即客户机) 请求数据或动作。另一端 (即服务器) 完成该请求。请求者与请求完成者的角色在许多不同的软件应用中经常能够发现。在操作系统级别使用客户机/服务器术语来描述进程间可能发生的生产者-消费者关系。例如, 当用一个命名管道连接两个进程时, 创建管道的进程通常称做服务器, 而请求此管道的进程通常称做客户机。

动态数据交换 (dynamic data exchange, DDE) 协议是另一个客户机/服务器范例的操作环境的例子。动态数据交换使用消息传递、共享内存、事务协议、同步规则以及会话协议来允许数据和控制信息在进程间流动。服务器对来自客户机的数据或动作请求作出反应。客户机和服务器可以按多种关系进行通信。单个 DDE 代理可能既是客户机, 也是服务器。也就是说, 进程可能从一个 DDE 代理请求服务的同时, 还为另一个 DDE 执行服务。当服务器需要请求信息或动作时, 它就变成了一个客户机。当客户机信息或动作接收请求时, 它就变成了一个服务器。一个服务器可以与任意数量的客户机进行通信。一个客户机也可以与任意数量的服务器进行通信。客户机/服务器的 DDE 形式通过操作系统提供的消息系统来完成。

也许最常用的客户机/服务器架构形式出现在网络环境中。使用客户机/服务器模型有各种不同的方式。表 9-1 列出了网络环境中一些常用的客户机/服务器模型应用。

表 9-1 网络环境中一些常用的客户机/服务器模型应用

客户机/服务器模型应用	描 述
文件服务器	充当共享数据库、文档、多媒体文件等的中心仓库。客户机通常是网络上的工作站或终端。客户机请求使用文件内的文件或记录。文件服务器将请求传输给客户机。服务器的任务之一是维护数据完整性并加强文件访问的安全性。
数据库服务器	分解网络环境中不同机器间应用的处理过程。客户机请求某特定的信息或数据项。数据库服务器将发现被请求的信息或数据。只提交客户机的请求。数据库服务器可以接收要求服务器多个数据库并集和交集的复杂信息查询。合并或相交数据库后, 数据库服务器然后代表客户机查找数据。
事务服务器	事务发生在包含事务服务器的一台或多台机器上。每个动作和更新必须在不被打断的情况下完整完成。如果事务碰到问题, 必须取消所有的动作和更新, 同时再次尝试该事务。
应用服务器	用于为多个客户机提供对应用的访问。应用服务器在服务器和客户机之间分解应用中的工作。尽管主要工作仍然在服务器上完成, 但客户机通常拥有自己的处理器来执行部分工作。

续表

客户机/服务器模型应用	描 述
逻辑服务器	用于解决那些需要进行大量符号计算的问题。逻辑服务器既可以查找数据库中的显式信息,也可以查找隐含信息。它可以从数据库推理或演绎出没有显式进入数据库的信息。逻辑服务器包含一个具有一个或多个内置推理引擎的数据库,该引擎用于从服务器获取结论和推论。数据库包含规则、公理、定理以及过程。逻辑服务器的每次查询都将导致逻辑服务器推理引擎执行演绎、归纳、推测,或一些组合动作

9.2.1 文件服务器

文件服务器 (file server) 是客户机/服务器模型的最基本应用之一。文件服务器充当共享数据库、文档、多媒体文件等的中心仓库。客户机通常是请求使用文件或文件中记录的网络上的工作站或终端。文件服务器然后将请求转发给服务器。服务器的任务之一是保持数据的完整性,并强制实施文件访问的安全性。互斥量和信号量用于防止竞争条件和死锁。文件服务器范例各式各样。使用客户机/服务器模型的文件服务的常用形式之一是 FTP (File Transfer Protocol, 文件传输协议), 它是文件传输的一般性协议。一般情况下, 有一台主机充当文件服务器, 允许远程用户登陆和执行文件传输 (上载或下载文件)。在这种配置中, 主机为服务器, 远程计算机为客户机。客户机向服务器请求发送或接收一个或多个文件, 服务器要么接收传输的文件, 要么传输被请求的文件作为应答。与基本文件传输请求一起, FTP 服务器常常还支持大量来自客户机的请求。表 9-2 列出了 FTP 对话期间, 客户机可能发出的基本命令。

表 9-2 FTP 对话期间客户机可能发出的基本命令

FTP 命令	描 述
!command	在本地主机上执行 command
append LocalFile RemoteFile	在远程文件 RemoteFile 上附加本地文件 LocalFile
bell	每次文件传输后响铃
bye	关闭当前远程主机连接并放弃 FTP
cd RemoteDirectory	将当前远程工作目录更改为 RemoteDirectory
close	关闭当前远程主机连接
delete RemoteFile	从远程主机删除远程文件 RemoteFile
get RemoteFile [LocalFile]	将远程文件 RemoteFile 复制到名叫 LocalFile 的本地文件。如果省略 LocalFile, 则使用 RemoteFile 名字
help [Command]	显示关于 Command 命令的帮助。如果忽略命令, 则显示所有 FTP 命令的列表

续表

FTP 命令	描 述
lcd LocalDirectory	将当前本地工作目录更改为本地目录 LocalDirectory
ls RemoteDirectory	列出当前远程目录 RemoteDirectory 的内容
mkdir RemoteDirectory	创建一个名叫 RemoteDirectory 的远程目录
open HostName [port]	试图连接到主机 HostName。如果指定了端口, FTP 假定端口为一个 FTP 服务器
put LocalFile [RemoteFile]	将本地文件 LocalFile 复制到远程文件 RemoteFile。如果没有给出远程文件, 则使用 RemoteFile 名字
pwd	显示当前远程工作目录
quit	关闭当前远程主机连接并放弃 FTP
rename RemoteFrom Remote To	将一个远程文件从 RemoteFrom 重命名为 RemoteTo
rmdir RemoteDirectory	删除远程目录 RemoteDirectory

线程处理 FTP 客户机/服务器

人们并不希望 FTP 主机按序列化方式来工作。当多个客户机请求文件传输时, 尽可能对每一个客户机作出反应对于 FTP 服务器来说很重要。如果主机一次只响应一个文件传输请求, 客户机将处于时间不确定的等待之中。在这里, 我们可以在客户机/服务器架构中应用多线程技术。我们这样设计服务器, 让一个单独的线程负责一个文件传输请求。客户机请求文件传输、服务器接受请求、创建线程处理请求以及等待下一个请求。图 9-3 显示了多线程处理的 FTP 服务器、请求以及客户机间的关系。

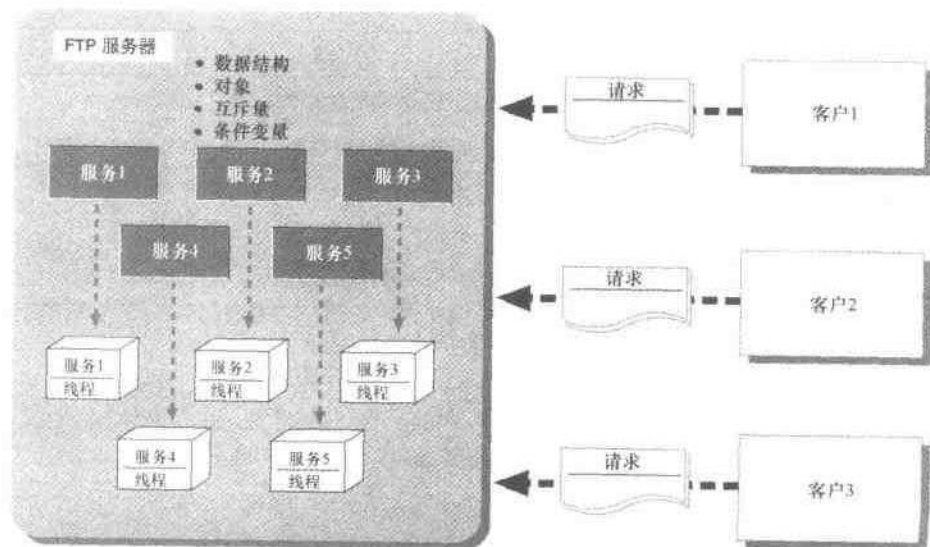


图 9-3 多线程处理的 FTP 服务器、请求以及客户机间的关系。服务器使用多个线程或多个进程来实现

请注意，这个客户机/服务器架构应用的本质要求服务器使用多个进程或多个线程来实现。因为线程更容易创建和管理，而且要求开销较少，所以，我们可以通过多线程化服务器来改进客户机/服务器架构。这是多线程处理的一个自然应用。客户机/服务器范例通常是一种多对一的关系。也就是说，通常会有多个客户机同时向一个服务器提出请求。通过让服务在一个单独的线程中为每个客户机提供服务，所执行工作模式的逻辑被大大简化了。多线程处理FTP服务器的基本架构如图9-4所示。

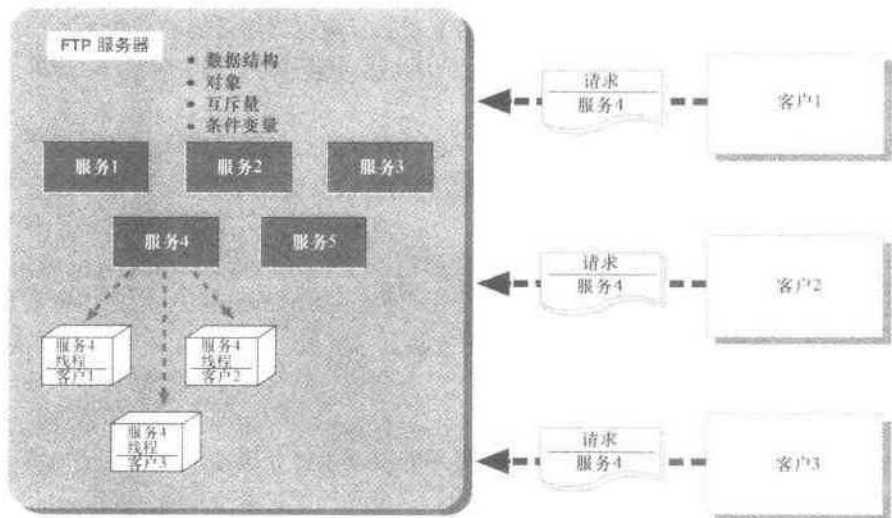


图 9-4 演示单个服务可能有多个激活线程，每个线程服务于一个不同的客户机

这里将服务器分成多个线程是有意义的。服务器可以看作为客户机提供的服务集合。每次客户机请求服务时，服务器分派一个线程来满足客户机的请求。图9-4显示单个服务可能有多个激活的线程，每个线程服务于一个不同的客户机。如果一台服务器有 N 个服务以及 M 个客户机，每个客户机使用一个唯一的服务，则服务器分派的线程数量为 $N \times M$ 。在这种情况下，线程间的关系容易理解，也易于维护和改进。客户机和服务器线程间关系的结构易于理解和通信。使用互斥量和条件变量是该架构的自然体现。

9.2.2 数据库服务器和事务服务器

数据库服务器通常以其组成客户机/服务器范例的技术而著称。数据库服务器是成功将应用程序处理过程分派到网络环境中不同机器上的初始应用之一。当使用一个文件服务器时，客户机通常发出文件请求，或某文件中一些记录的请求，然后服务器通过网络发送文件或记录给客户机来作出响应。然后，客户机进行文件或记录的余下处理。数据库服务器执行方式与之稍有不同。一般情况下，客户机请求一个信息或数据项。数据库服务器不是发送整个数据库给客户机并要求客户机在数据库中查找数据，数据库服务器将完成需求数据或信息的查找。然后，它只提交客户机请求。数据库服务器比文件服务器完成的工作要多。通常，文件服务器不关心信息如何被使用，或者需要文件中的什么特定信息。它只是简单地接收客户机的请求，并发送所请求的文件。另一

方面，数据库服务器通常接收复杂信息查询，它要求服务器联合和交叉查找多个数据库。一旦联合和相交多个数据库，数据库服务器就代表客户机搜索数据。无论找到数据与否，数据库服务器只将反应发送给客户机。

与文件服务器一样，数据库服务器也可以通过多线程架构来实现。数据库服务器可能接受的查询的复杂性要求服务器或者派生子进程来负责每个请求，或者服务器分派一个单独的线程来负责每个请求。假设一下，如果一个客户机由于一个复杂的查询占据着数据库服务的资源，其它客户机列表必须一直等待。这是一种不可接受的事务状态。数据库服务器的架构是另一个具有内在并发性的明显例子。一般而言，在客户机/服务器范例中，服务器或使用多个进程服务于每个请求来实现，或者使用多个线程来服务于每个请求。

虽然我们着重讨论了发生于服务器的处理过程，但客户机并不是毫无责任。发送到数据库服务器的请求通常是以 SQL 或其它数据库查询语言的形式发送的。在发送到数据库服务器之前必须首先生成请求。这通常属于客户机软件的工作之一。如果客户机运行交互式软件，客户机要负责管理数据项、错误处理以及结果显示。这部分工作通常涉及到图形用户界面 (graphical user interface, GUI)，它显示给用户输入屏幕和输出屏幕。图形用户界面让用户通过构建数据库查询与客户机软件交互。只有构建一条查询后，才能向数据库服务器发出请求。数据库服务器仅返回查询的结果。它不负责为客户机格式化查询。客户机软件的用户界面也自然符合于线程处理。同样，我们让架构来决定哪些地方需要多线程处理。图形用户界面一般使用事件驱动范例来构建。我们将看到，这个模型通常包括消息队列、应用窗口、消息处理器以及事件处理器。可以方便地将这些组件的不同组合分配给单独的线程。图 9-5 显示了数据库/服务器模型的新一般性架构。

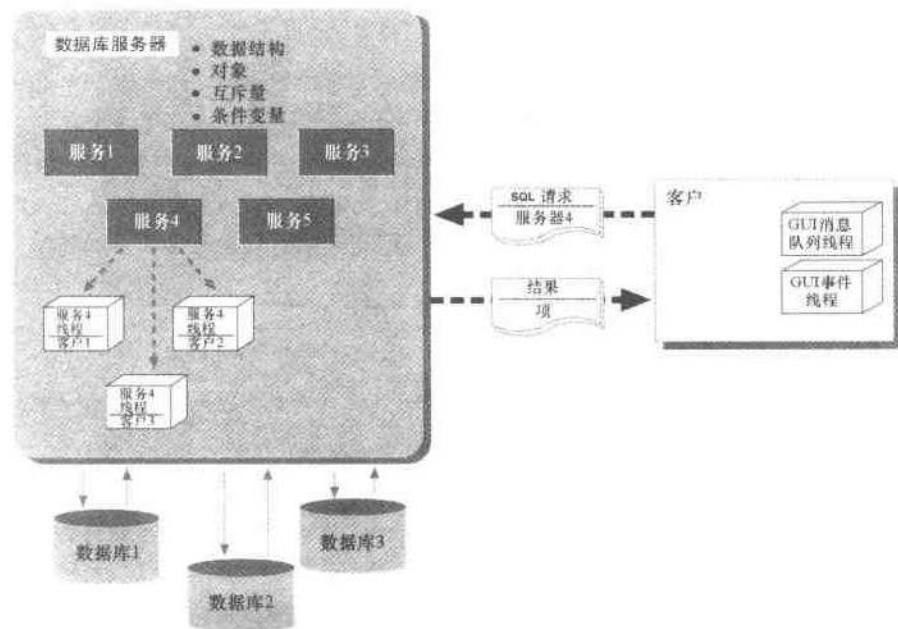


图 9-5 数据库/服务器模型的一般性架构

我们可以扩展数据库/服务器范例的一般性架构，花极少气力具有事件驱动 GUI 的客户机就可以在 GUI 代码架构内利用多线程处理技术。

数据库服务器模型可以扩展到包含事务服务器 (transaction server)。事务是善于简单数据库查询的一个分支。事务是一个动作和更新序列，它必须应用于一个或多个数据库。事务发生于包含事务服务器的一台或多台机器上。第一条技巧是这些动作和更新必须通过第8章所讨论的 ACID (原子性、一致性、孤立性以及持久性) 测试。为了让事务成功，每个动作和更新必须完整执行，而不能被打断。无论事务碰到什么样的问题，更新和动作必须取消。数据库为必须回置到事务开始前的状态，而且事务必须重新尝试。前面讲过，在这样的条件下，使用事件互斥量来帮助程序员设计一个可以通常 ACID 测试的事务。数据库服务器相对于文件服务器来讲，它为客户机所做的工作要多一些。事务服务器为客户机所做的工作又比数据库服务器要多。不过，随着服务器工作负荷的增加，客户机端通常需要做更多的工作。这是因为当我们从文件服务器转换到数据库服务器时，请求的组合增加了难度。这就带来了客户机和服务器的一个重要指派方法。

胖和瘦名字调用

服务器分为瘦服务器 (thin server) 和胖服务器 (fat server)。客户机/服务器范例中的瘦服务器使客户机承担大部分工作。文件服务器就是一个瘦服务器的典型例子。文件服务器只给客户机发送被请求的文件，需要对文件所作的一切处理工作都由客户机负责。另一方面，胖服务器意味着它在客户机/服务器请求中承担大部分工作。事务服务器就是胖服务器的典型例子。客户机请求要完成的事务，然后交给服务器去做大部分工作。与存在胖服务器与瘦服务器类似，客户机也存在胖客户机和瘦客户机。通常而言，“瘦”一词并不用于服务器，相反，这一词常用于客户机端，同时我们还可以讲胖客户机。在数据库客户机/服务器环境中，客户机有一个 GUI 或语音激活用户接口，客户机通常完成大部分工作。尽管数据库服务器上的处理任务相当多，但为了产生实际的请求，图形处理和语音识别可能需要更多的处理工作。在这些条件下，客户机就被看作是“胖”客户机。客户机/服务器范例中的服务器通常需要多线程化或通过多进程来构建。只要我们拥有了一个胖客户机，客户机的“肥胖”程度通常暗示了多线程处理技术的使用。所以，当我们考察一个客户机/服务器架构的时候，在应用多个线程的地方，服务器和胖客户机都是理想之选。

9.2.3 应用服务器

应用服务器用于将软件片断完成的工作分解成两个或多个进程 (或线程)。至少其中一个进程为服务器，其中一个进程为客户机。应用服务器可能比数据库服务器和事务服务器更复杂。

数据库或事务服务器的焦点是为客户机提供访问一个 (或一套) 数据库的能力，而应用服务器用于为多个客户机提供对应应用的访问能力。这是来源于大型机 (mainframe) 世界旧式主机-远程端 (host-remote) 模式的变种。在主机-远程端范例中，程序在一个称作主机 (host) 的计算机上寄居并执行。通过通信线路连接到主机的远程终端访问运行于主机上的程序。一般情况下，远程终端是一个哑终端 (dumb terminal)。也就是说，终端不包含本地处理器。在主机-远程端范例中，远程终端承担的任务很少。主机计算机囊括所有的处理任务，而且负责完整执行程序。远程终端的主要责任是作为一个输入和显示设备。按这种方式，主机计算机充当所有相连远程设备的应用服务器。

一般来说,应用服务器范例将应用中的工作分配到服务器和客户机上。虽然主要工作仍然由服务器承担,但客户机通常有自己的处理器,同时执行部分工作。例如,分布式专家系统(distributed expert system)就是如此。图 9-6 是一个基本分布式专家系统的条块图。服务器包含知识库、系统状态和推理引擎。客户机包含用户接口处理、解释子系统以及知识获取子系统。

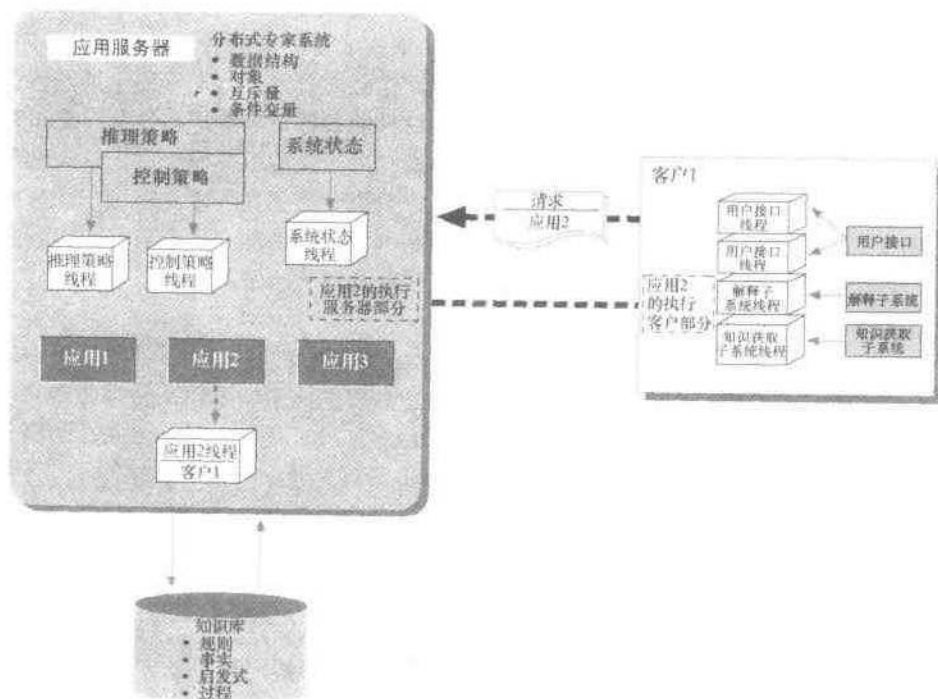


图 9-6 基本分布式专家系统的条块图。服务器将访问知识库,同时包含系统状态和推理引擎。

客户机包含用户接口处理、解释子系统以及知识获取了系统

知识库类似于数据库,其中包含大量存储信息。不同之处是,知识库包含的不仅仅是简单的事实或信息块。知识库包含规则、推断法、陈述性知识以及与特定域对应的过程化知识,还加上一些简单的事实。按数据库查询从数据库中获取信息的同样方式,使用推理引擎从知识库获取事实、知识、结论以及推论。

专家系统的每条请求都导致对知识的搜索。专家系统中所用的搜索通常比数据库系统中查找数据所用的搜索更复杂。

知识库搜索可能也把问题解决或其它复杂处理过程作为搜索的一部分。例如,对于一个司法专家系统,它设计用来匹配取样指印与文件上已经记录的指印。完整的指印模式保存在知识库中。取样指印搜索的目标是在与取样指印最接近的知识库中找出完整的指印。为了让搜索成功,专家系统必须执行模式匹配、边缘检测、高空间过滤、低空间过滤以及其它不同类型的取样指印图像处理。图像处理要耗费大量的 CPU 时间,它涉及冗长的计算。当使用知识库时,这类处理就被视作搜索的常规部分。就像数据为服务器的每次搜索由服务器中的一个单独线程来完成一样,专家

系统服务器也分派一个单独的线程来处理每个请求。

与文件服务器、数据库服务器和事务服务器相反，应用服务器通常既有胖客户机，也有胖服务器。请注意，图 9-6 中客户机的责任包括几个部分。每个部分在自己的进程或线程内执行。用户接口组件通常分派于多个线程中。专家系统应用服务器的架构与其它客户机/服务器实现一致，对服务器的每个请求都应当由一个单独的线程来处理。客户机端的用户接口也应当分配给多个线程。

当选择应用服务器来实现客户机/服务器关系时必须小心。应用的架构必须明确定义，能够准确理解。而传统应用设计执行于单台计算机上，应用服务器代表一种分布式应用，它的一部分工作在服务器上执行，一部分在客户机上执行。理想情况下，应用应当分解成大体独立的多个组件。具有内在并发性的组件通过多线程来实现。具有分离关系的组件也可以通过多线程来实现。当选定适当的软件结构后，多线程技术的应用就显而易见了。

逻辑服务器

逻辑服务器是一种特殊类型的应用服务器，用于需要大量符号计算的问题解决方面。例如，语音识别、模型识别、图像处理以及自然语言处理都比传统数据库处理需要更多不同类型的计算。相对于传统数据库服务器而言，逻辑服务器的主要优点是，它既能从数据库查找显式信息，还能查找隐含信息。逻辑服务器能够从数据库信息进行演绎和推理，得出的信息是数据库没有显式纳入的。数据库服务器只返回数据库中那些显而易见的可用信息，但逻辑服务器基于数据库已经存在的信息，它可以得到新的信息。

逻辑服务器由包含一个或多个内置推理引擎的数据库组成，这些推理引擎用于从服务器获取结论与推论。逻辑服务器数据库包括规则、公理、定理以及过程组成。逻辑服务器的每次查询都将使逻辑服务器推理引擎进行演绎、归纳、推测或兼而有之。传统数据库查询让数据库使用索引(index)、记录(record)、键(key)进行搜索，但逻辑服务器中的数据库查询导致自动推理过程的发生。自动推理通常通过图解搜索算法(graph search algorithm)来实现。许多自动推理都使用宽度优先(breadth-first)和深度优先(depth-first)搜索图解算法来实现。这些图解算法用于模拟演绎、归纳和推测的过程。

例如，我可以建立一个简单的逻辑服务器，让它回答有关网络上的数据路径问题。我们可以建立一个数据库，其中每条记录由两个字段组成：原始节点和目标节点。每条记录表示从原始节点到目标节点的一条路径。然后向服务器查询从一个工作站到另一个工作站的可能路径。假如数据库包含 4 条记录：

记录 1：工作站 A，工作站 D

记录 2：工作站 F，工作站 B

记录 3：工作站 B，工作站 A

记录 4：工作站 E，工作站 F

客户机可以查询从工作站 E 到工作站 A 的路径。如果我们使用的是传统的数据库搜索技术，搜索完这 4 条记录后，它将返回没有从工作站 E 到工作站 A 路径的信息。不过，使用逻辑服务器中的技术，就可以找到一条从工作站 E 到工作站 A 的路径。这条路径是演绎推理的结果。推理过程如下：

存在一条从工作站 E 到工作站 A 的路径。
 存在一条从工作站 F 到工作站 B 的路径。
 存在一条从工作站 B 到工作站 A 的路径。
 所以, 也应存在一条从工作站 E 到工作站 A 的路径。
 这条路径就是:

$E \rightarrow F$

$F \rightarrow B$

$B \rightarrow A$

通过深度优先图解遍历算法, 或宽度优先图解遍历算法很容易实现以上演绎推理。注意, 找到这条路径只要求判断 4 个假设, 并且作出一个演绎式推理。每条记录表示一个假设, 演绎步骤为: 所以, 也应存在一条从工作站 E 到工作站 A 的路径。

在实际中, 逻辑服务器可能要考虑上千个假设, 并作出上千个演绎推理, 才能为客户机提供相应的答案。考虑假设与逻辑服务器部分的推理组成了推论链。每个推论链都可以作为一个单独的线程。逻辑服务器的性能通常以 LIPS (logical inferences per second, 每秒逻辑推理数) 来衡量。

与其它类型的服务器一样, 逻辑服务器也是使用客户机/服务器范例来实现的。图 9-7 显示了多线程逻辑服务器架构的基本结构。

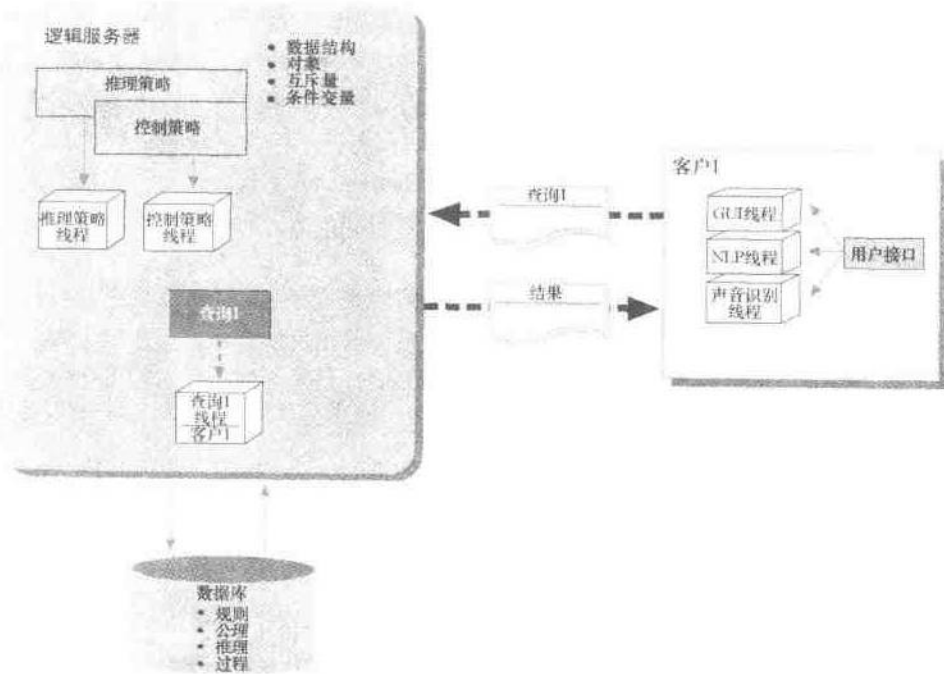


图 9-7 多线程逻辑服务器架构的基本结构

客户机软件的主要责任是按适当的格式格式化查询，以便逻辑服务器的使用。逻辑服务器可能通过语音激活，可能需要自然语言接口。对于这种情况，客户机将是一个胖客户机。逻辑服务器总是一个胖服务器。如果服务器端需要简单处理过程，则可以使用传统的数据库技术。鉴于组成大部分逻辑服务器处理过程的推理引擎的本质，将服务器分解成多个线程也是自然之举。

9.2.4 事件驱动架构

另一个可用于多线程架构基本的编程模型是事件驱动模型（event-driven model）。在事件驱动模型中有 5 个基本组件：

- 事件循环；
- 事件队列；
- 事件；
- 事件处理器；
- 窗口。

图 9-8 显示了这些组件间的基本关系。事件队列从操作环境接收事件。

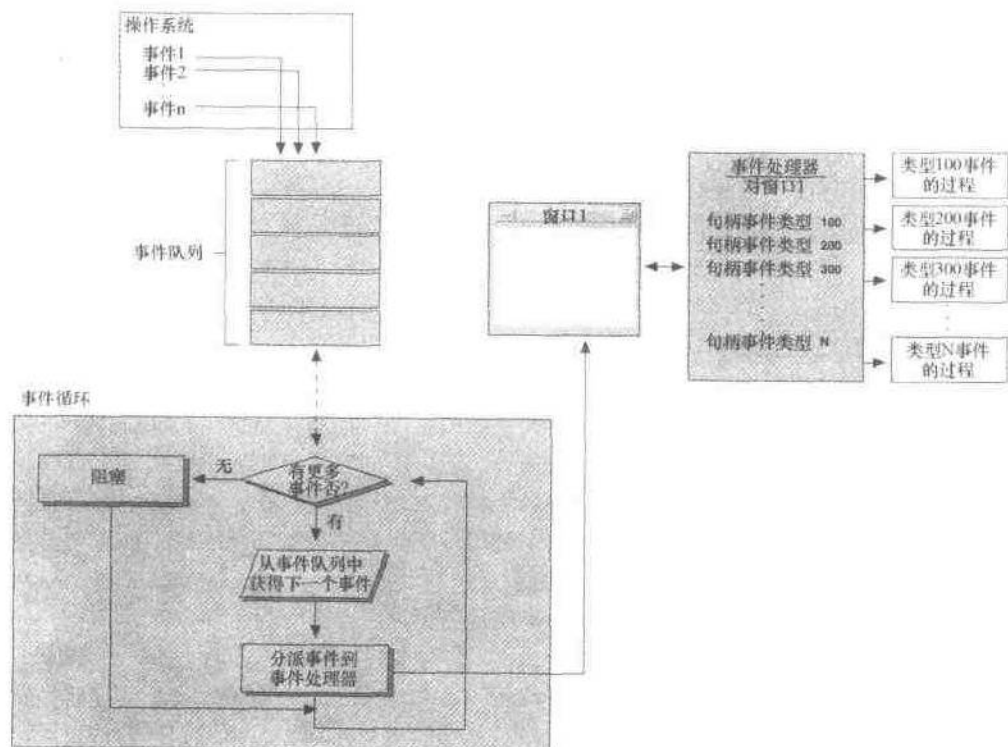


图 9-8 事件循环、事件队列、事件、事件处理器以及窗口组件间的基本关系

事件循环不断从事件队列中读取事件，并将这些事件分派到适当的窗口。每个窗口都有一个

相关联的事件处理器 (event handler)。事件处理器为窗口可能接收的每种类型的事件定义了处理方式。事件驱动模型有时也称做消息模型 (message model)，它将每条消息或每个事件指派一种类型。事件或消息处理器基于类型来处理消息。例如，我们可以用一个整数来表示窗口接收的消息。而且，我们可以将 100 指定为打印请求，将 200 指定为排序请求，将 400 指定为计算请求。然后建立消息处理器来完成某种类型的处理过程，如果接收的消息是 100，则完成与之对应的处理过程，如果接收消息是 200，则完成另一种处理过程，如果接收的消息是 400，则完成的处理过程又与之不同。

循环分派消息，消息处理器基于所接收的消息进行处理。因为消息或事件处理器被分成了多个程序 (routine)，其中一些程序需要较长时间的处理过程，所以事件驱动模型是多线程处理的另一个自然之选。多线程事件驱动模型的另一个典型例子是 Windows NT 和 OS/2 这样环境中可以发现的 GUI。在这些环境中，每个应用程序至少包含一个线程，它称做主线程 (main thread)。主线程包括：

- 至少一个窗口 (可能不可见)；
- 一个消息队列；
- 一个消息循环；
- 一个消息处理器。

程序员可能给单个消息队列分配多个窗口，消息处理器由主线程来处理。如果一些消息需要长时间的处理，这可能会导致问题的发生。如果主线程包含 3 个窗口，其中窗口 1 获得一条需要长时间处理的消息，发送到窗口 1 和窗口 3 的消息就会不必要地被延迟。为了避免不必要的延迟，提高应用程序的反应速度，程序员可以创建第二个线程。第二个线程有自己的消息队列、消息处理器以及窗口。这样，应用程序可以利用该环境的多线程功能了。图 9-9 显示了多线程事件模型的结构。

事件驱动模型中的第一级多线程分解发生在消息队列以及窗口级别。程序员可以引入许多线程，每个线程都拥有自己的消息队列、消息处理器和窗口等等。这为应用程序提供了基本级别的多线程处理。事件驱动模型内的第二级多线程处理发生在事件或消息处理器中。任何需要长时间处理的事件或消息应当分配一个单独的线程。许多应用程序自动将打印请求、文件排序、文件搜索以及长时间计算分配给单独线程。由于这些任务项能由不同的事件来表示，所以事件驱动模型在这种情况下能正常工作。程序员可以将这些事件建立为一套分离程序 (routine)，它们之间很少或根本没有交互。分离处理通常是在应用程序中利用多线程技术的最简单途径。如果两个进程没有关系，而且不共享任何临界区 (critical section) 或公共资源 (common resource)，那么这两个进程是分离的。通过分离程序，如果需要使用同步技术的话，也不会使用太多。

客户机/服务器模型可以将应用分解为服务器和客户机组件，服务器组件分解为多线程，每个线程处理不同的服务，事件驱动模型将应用分解成一系列多事件队列、事件消息处理器以及窗口 (分配给消息处理器)，将事件队列和处理器分配给单独的线程。同样，消息处理器也可以进一步分解为多个线程，其中给需要长处理时间的消息分配自己的线程。事件驱动模型也有一些内在的并发性质。我们可以充分利用这种内在的并发性与多线程。

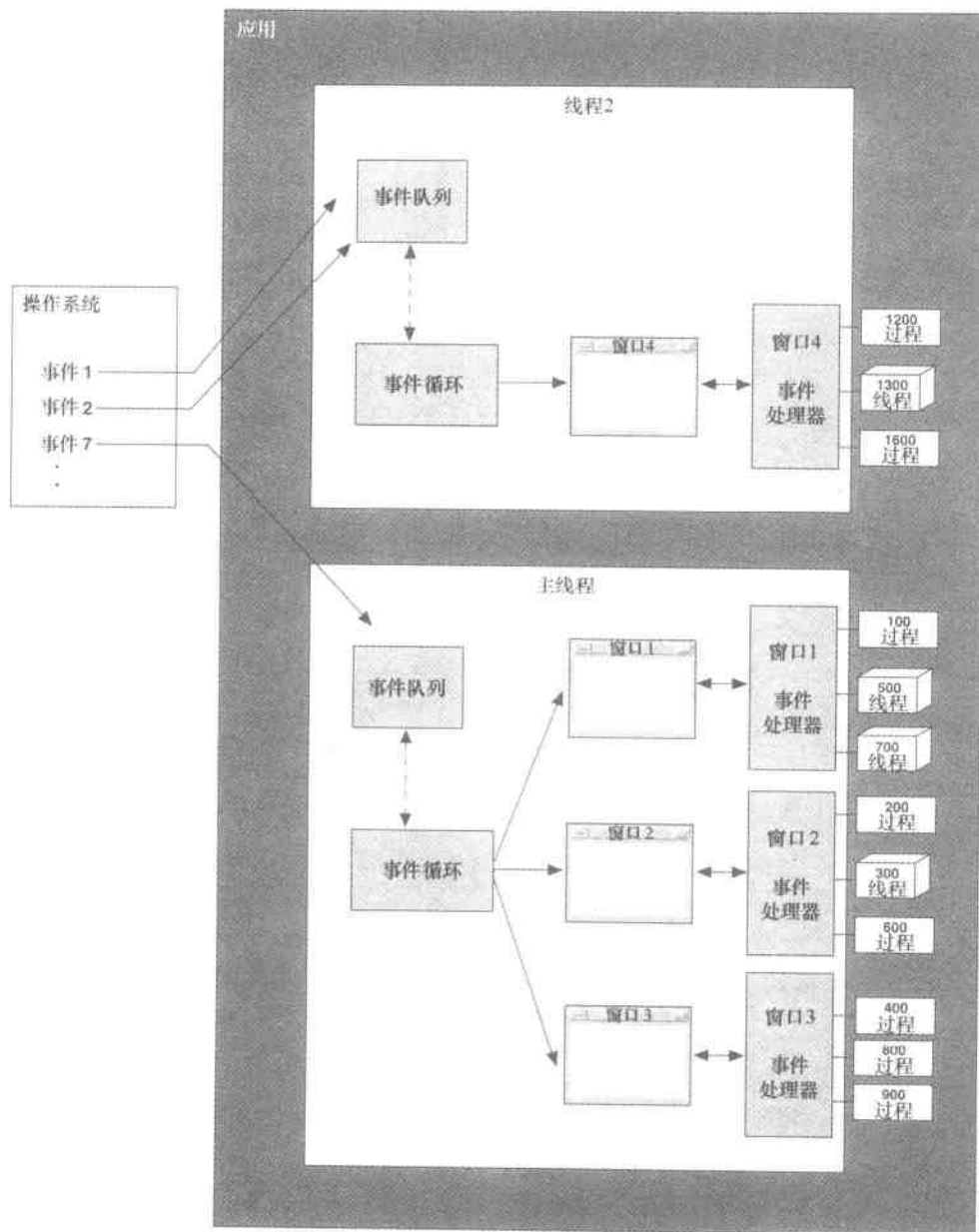


图 9-9 多线程事件模型的结构

9.3 黑板架构

黑板（blackboard）是一种问题解决结构。与专家系统一样，黑板设计用于解决特定的问题。

也就是说，它们不是用于多种目的的应用程序，例如既作为字处理器、电子表格或 DBS。实现黑板来解决非常具体的问题，生成的解决方案也只针对该问题才有用。已经实现的一些重要黑板有笔迹分析、化学鉴定、语音识别、声纳分析、图像处理、任务鉴定规划、信号-符号的转换、心理测试、符号积分、符号微分以及自然语言处理。特殊的计算语言或操作环境与结构解决问题的途径相比，后者更是黑板的特色所在。

黑板架构由两个基本组件构成：

- 黑板数据结构；
- 知识源。

通过知识源，在问题解决进程中协作使用黑板。知识源是一个具有特定域过程化知识、陈述性知识或两者兼有的软件组件。知识源也包含一个或多个推理引擎，它可以工作于所包含的过程化或陈述性知识之上。为了弄清楚知识源和黑板数据结构的意义，假设我们碰到了这样一位旅游者，他来自于一个远方国度，他所讲的语言是我们从未听过的。我们能求助的工具只有一台笔记本电脑，它配备了一个能专业识别语言的黑板。陌生人发出声音，我们根据黑板软件将旅游者的话翻译过来，我们才能作出适当的反应。根据 Avron Barr 和 Feigenbaum 的观点，对于 7 个重要语言学方面的学习可以帮助我们翻译陌生人所讲的话 (Barr, Feigenbaum, 1981)。表 9-3 列出了这 7 个方面及其描述。

表 9-3 7 个语言学方面及其描述

语言学方面	描 述
语音学	表示词汇中所有单词的发音物理特征
音位学	描述单词在句子中发声时语音上出现变化的规则
语素学	描述如何将意义单元（称做词素）组合成单词的规则
韵律学	描述句子中重音和语调变化的规则
语法	语法或造句规则
语义	单词和句子的意义
语用学	对话规则。处理语言与上下文的关系

在黑板模型中，对于 7 个语言学方面，每个方面都配备了一个知识源软件组件。外国人发出的原始语句将记录在黑板上。然后每个知识源尝试应用它的专家知识翻译陌生人所讲的话。知识源可以同步和并发操作。不存在集序 (set order)。知识源生成与被翻译语句相关的假设或输入后，知识源的输入也将被放在黑板上。这样，黑板包含知识源合作产生的当前工作假设。如果一条假设不正确，它就会被另一个更适合于这种情况的假设所抢占。基于黑板上的内容以及监控是否生成可接受解决方案的控制机制触发每个知识源中的有用知识。最终，语义学 (semantics) 知识源和语用学 (pragmatics) 知识源将判断陌生人话语的正确意义，然后，我们调用翻译器将它转换成我们可以理解的语言。知识源于是充当智能代理，这些智能代理被灌输许多领域的专家知识，黑板是当前最佳假设、要解决的问题以及所需求解决方案的仓库。

从知识源间的关系可见，黑板模型是多线程技术的理想之选。黑板的临界区可以受互斥量和条件变量的保护。可以将每个知识源分配给一个单独的线程。图 9-10 显示了多线程处理黑板架构的基本结构。

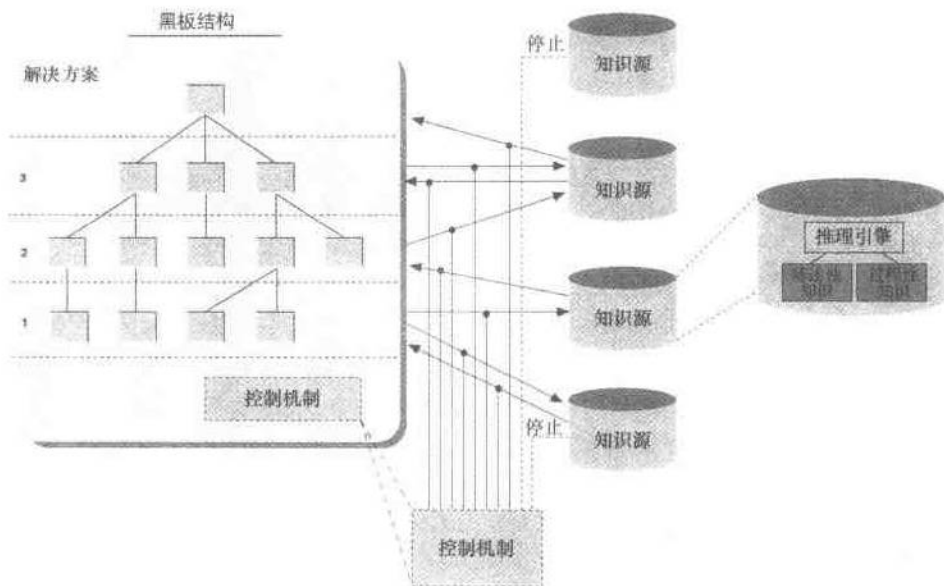


图 9-10 多线程处理黑板架构的基本结构

知识源所执行的处理过程可能要消耗很多的处理器时间。就像在逻辑服务器中一样，知识源可以执行演绎、归纳或推测。这种推理可能包含上千次推论与假设过程。我们并不希望序列化在这种条件下知识源所做的处理过程。特别是，一个知识源的输入可能是另一个知识源的输出。它们经常重复和递归性地一起工作。有时，它们也可能以一个分离的方式工作。不过，唯一发生的交互就是通过黑板数据结构，它是一个临界区。明白这一点后，我们就知道黑板模型是如何适宜用于多线程处理。

架构基础

客户机/服务器、事件驱动以及黑板架构都适宜于多线程处理。它们的意义明确。它们的结构中具有内在的并发性。而且，它们可以混合和匹配形成更可靠的架构。我们可以得到客户/服务器，其中的客户为事件驱动，服务器通过黑板结构来实现。这 3 个架构是程序员编写多线程应用程序的良好起始基础。选择适当的架构基础是所谓增量多线程处理概念的开始。选定一种适合于多线程处理的架构后，我们就可以进一步选择和构建用于实现这些多线程架构的 C++ 组件。

9.4 途径上的不同（面向对象与过程化）

虽然以上讨论的架构适应于多线程处理，但它们通常使用非面向对象技术来实现。这些架构

传统上使用过程化编程方式（procedural style of programming）来实现。在这些模型中实现多线程处理技术，当应用于中型到大型软件开发中时，就会暴露出过程化编程方式固有的弱点。过程化编程方式的特点是在系统开发中使用自顶向下（top-down）、逐步求精法（stepwise refinement approach）。在自顶向下方式中，系统或应用程序用单个主过程来描述。然后将这个过程分解成逻辑子过程（logical subprocedure）。这些逻辑子过程被进一步分解成逻辑子过程。这个分解过程一直持续到整个应用程序能够在程序语句级进行描述。在过程化编程方式中，将应用程序看作过程和函数的集合。

例如，有一个求数学表达式值的应用程序，其中的数学表达式以字符串的形式获取。主过程可为：Evaluate Expression，图 9-11 中的 VTOC（内容可视表）显示了典型的自顶向下途径来描述 Evaluate Expression 的子函数。

图 9-11 中的第一个块代表一个模块、过程或函数。通过这种方式描述了整个应用程序的自顶向下编程方式。程序员将应用程序分解为适当的函数，然后通过他所使用的计算机语言为函数编写代码。使用过程化编程方式时，数据被看作一个“二等公民”。在过程化编程中，在将应用程序分解为函数部分时，几乎不考虑所使用的数据结构和数据库。数据只被看作应用程序中过程和函数所使用的对象而已。应用程序以过程为中心。在多线程应用程序中的主要缺陷是竞争条件和死锁。竞争条件和死锁以数据和资源为焦点。正是临界区（数据）和资源（数据和设备）的误用才导致多线程应用程序出现问题。多线程的过程化方式因为是以函数为中心，而不是以数据为中心，所以存在缺陷。

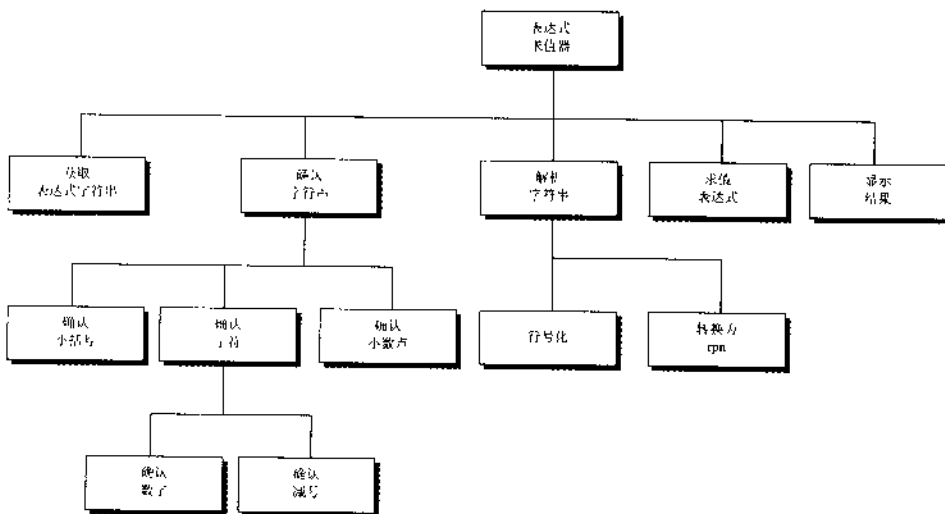


图 9-11 表达式求值程序的 VTOC（内容可视表）

过程化编程模型中的过程和函数只将数据看作传递的参数，或看作全局变量。数据不属于任何过程。作用域内的任何过程或函数都可以平等地使用（或滥用）它。临界区、互斥量和条件变量可以在过程、函数间自由传递，有时这种传递可能是无意中的进行的。这通常会导致过程化程序

中发生竞争条件和死锁。虽然我们可以选择适宜于多线程处理的架构基础,但常常由于使用了错误的实现技术,而抵消了该基础的优点。

在这种情况下,事实证明,在多线程应用程序中使用过程化编程得到的程序容易出错、难以理解和维护、架构脆弱。因为多线程应用程序的主要问题发生在临界区、共享资源的处理过程中,所以,我们需要一种与之不同、以数据为中心的途径来进行多线程处理。

9.4.1 封装是关键(保护和数据隐藏)

使用传统过程化编程构建多线程应用程序的一个主要问题是软件生命周期的确定性。软件开发中的一条谚语是:只有优秀软件片断才有用武之地。根据这条谚语可推知,任何被使用的软件最终都要被修改。在多线程应用程序中,如果使用过程化编程技术,就会给软件修改带来麻烦。

有用的软件必定要升级修改。这种修改可以发生在软件的任何地方。在软件更新中,曾经包含临界区的代码段可能修改后不再包含临界区。对于保护临界区的同步变量又将如何变化呢?如果临界区为同步和条件变量复杂网络的一部分,为了确保各个线程仍然协调一致,程序员该如何做呢?例如,有一个被4个线程并发访问的链表。使用同步技术来保护这个链表。如果修改程序,只让3个线程访问这个链表,那么整个同步方案可能要发生变化。同样,对某代码段进行修改,而此代码段最先并不包含临界区,更新后可能引入一个未保护的临界区。例如,如果线程A是唯一更新变量Count的线程,我们添加一个新线程(线程B),线程B与线程A并发执行,而且线程B也修改变量Count。如果程序员没有意识到创建好一个新临界区,就会存在竞争条件。

当添加新模块时,还带来了另一个缺陷:不知不觉访问现在临界区。发生的原因有多种,主要原因是,多线程应用程序内的任何变量都是潜在的临界区,而且没有可以强制实施的途径来标识哪一个变量正被多个线程使用。而且,也没有办法辨别程序中的哪一个变量将在临界区中使用。在不断修改和升级的系统或应用程序中,没有办法预先知道将来的线程将选定哪一个变量来访问,并让这个变量成为临界区。请记住,使用互斥量来保护临界区是可选的,并不是强制性的。就像在C++中,没有哪一种语言结构或规则标识哪个变量为临界区,没有规则或语言结构强迫程序员必须在应用程序中保护临界区。互斥量的成功使用取决于程序员在访问临界区的地方协调使用互斥量。如果程序员没有意识到临界区正被操纵,他就不会去锁定互斥量或测试条件变量。程序员可能无意识地声明了一个可能隐藏的新同步变量,于是有效地禁止了原始同步变量。随着系统变得越来越大,越来越复杂,管理临界区和防止死锁也变得越来越困难。因为自顶向下设计模型是以过程为中心,临界区以数据为中心,所以它们并不匹配。在中型到大型的多线程软件开发工作中,临界区为移动的目标,以过程为中心的编程技术将无能为力。

在多线程应用程序中,任何非常量数据都有可能成为一个临界区。在中型到大型的多线程应用程序中,如何才能成功管理临界区以及其它共享资源呢?显然,笔者在此已经暗示了答案。我们的经验是,可使用面向对象编程技术来解决多线程应用程序中大部分管理临界区及其它共享资源的问题。我们建议,适当使用封装来组合互斥类和事件类与即将在多线程环境中使用的所有类,这是控制竞争条件的最有效可行方法之一。

在具有面向对象架构的应用程序中,应用程序的数据封装在类结构中。对这些数据的访问由类成员函数控制。类访问数据的唯一方式是通过类的接口。类发生变化,类的接口也将发生变化。

如果类的接口在某处被修改，它在所有地方都会发生变化。当类的成员函数通过内部互斥量保护类的数据时，多线程应用程序的大小，或应用程序的变化速度就不再成为问题了。在具有面向架构的多线程应用程序中，所有的临界区都受相应类接口的控制。所有的资源由类来建模，所以也是受类接口的控制。再也不必像在多线程应用程序中一样需要想办法查找、预测或管理每个临界区，面向对象编程允许程序员在类中集中进行同步和保护工作。使用封装，程序员在一个地方保护和同步化数据，此同步化和保护将随类应用到所有的地方。因为具有面向对象架构的应用程序中所有的数据都被封装，因此所有的临界区也被封装。如果修改和访问类数据的成员函数提供了锁定，那么竞争条件——多线程编程的敌人——就会得到抑制。

一、封装在多线程应用程序中的两个重要用途

首先，我们必须使用 C++ 类 (class) 或结构 (struct) 这种结构在应用程序中封装数据。其次，直接关联同步对象或保护对象与进行同步或保护的對象。通过继承 (inheritance) 或者包容关系来完成。这意味着，只要有可能，我们就要避免使用自由漂浮的互斥量、信号量以及条件变量。还要避免使用那些可以被函数或过程 (具有作用域访问权) 访问的自由漂浮数据。相反，在多线程应用程序中使用的每个类都将包含自己的同步和保护机制。当我们结合同步对象与它保护的类时，就不存在如何或何处使用同步对象的问题了。类提供它自己的同步和保护策略。类封装它的互斥量和条件变量。所以，我们可以使用 C++ 的封装机制提高多线程应用程序的可靠性，其途径至少两种：

- 在 C++ 类中封装互斥量和条件变量，只提供成员函数的访问权限。
- 通过继承或复合结合互斥量或条件变量类与宿主类。

表达式求值程序的结构如图 9-11 所示。这个结构是一个典型地按自顶向下来设计软件计算器的方式。我们可以使用两种封装技术将基于过程的软件计算器结构转换成面向对象结构。首先为一些相应的线程 API 提供一个接口类。在这里，我们使用 POSIX 线程库 API 来实现一个简单的互斥类，如程序清单 9-1 所示。

程序清单 9-1 一个简单的互斥接口类

```
class mutex{
protected:
    pthread_mutex_t Mutex;
    general_exception SemException;
public:
    mutex(void);
    ~mutex(void);
    void lock(void);
    void unlock(void);
};

mutex::mutex(void)
{
    if(pthread_mutex_init(&Mutex,NULL)){
```

```

        SemException.message("Could Not Create Mutex");
        throw SemException;
    }
}

mutex::~mutex(void)
{
    if(pthread_mutex_destroy(&Mutex)){
        SemException.message("Could Not Destroy Mutex");
        throw SemException;
    }
}

void mutex::lock(void)
{
    pthread_mutex_lock(&Mutex);
}

void mutex::unlock(void)
{
    pthread_mutex_unlock(&Mutex);
}

```

接口类互斥量封装了来自 pthread API 的若干组件。类封装了 pthread_mutex_t Mutex。Mutex 只能通过互斥类的成员函数来访问。类还封装了 pthread_mutex_lock()、pthread_mutex_unlock() 以及 pthread_mutex_destroy() 函数。互斥接口类的生成版本可能要复杂得多。理想情况下, 接口类应当封装 pthread API 为互斥量提供的所有功能性。这个类一般要包括异常处理策略。我们举例说明这样一个简单的互斥接口类版本作为 C++ 程序员的一个起点。

为互斥量和条件变量设计和编写接口类代码是消除多线程应用程序中所有不必要、自由漂浮同步变量的第一步。实现互斥类后, 我结合互斥类与其宿主类, 这里宿主类是 mt_calculator 类。我们可以通过继承或复合来结合互斥量。在这里, 我们使用继承来结合互斥类与 mt_calculator 类:

```

class mt_calculator:private mutex{
private:
    calculator Calculator;
public:
    double evaluate(string Input);
    list<expression_component> parse(string Input)';
};

```

mt_calculator 类的类关系如图 9-12 所示。我们声明 mutex 和 Calculator 对象为 mt_calculator 类中的私有对象, 因为我们希望 mt_calculator 类为一个具体类。mt_calculator 类不打算作为一个基类。mt_calculator 类的结构演示了两种封装技术。首先, mutex 被封装在 C++ 类中。这使我们将对 mutex 的访问权限限制于类成员函数。按这种方式, 我们将完全控制 mutex 的使用方式及用途。第二, mutex 类与用于保护和同步的 mt_calculator 类直接关联。所以, 不存在互斥量应当保

护哪一个临界区的问题。我们知道，只是 `mt_calculator` 类中的临界区被这个互斥量保护和同步化。同步发生在 `mt_calculator` 类的两个成员函数中：

```
double mt_calculator::evaluate(string Input)
{
    double Temp;
    lock();
    Calculator.inputString(Input);
    Calculator.process();
    Temp=Calculator.result();
    unlock();
    return(Temp);
}

list<expression_component> mt_calculator::parse(string Input)
{
    list <expression_component> Temp;
    lock();
    Calculator.inputString(Input);
    Calculator.validate();
    Calculator.parse();
    Temp=Calculator.expressionToken();
    unlock();
    return(Temp);
}
```

`evaluate()`和`parse()`成员函数通过 `mt_calculator` 类从互斥类继承的 `lock()`和`unlock()`成员函数保护它们的临界区。因为 `mt_calculator` 类的临界区受一个互斥量的保护，所以 `mt_calculator` 类可以安全地被多个线程使用。图 9-12 中的类关系演示了构建多线程化类时可以使用的其中一种基本技术。图 9-12 中的 `mt_calculator` 类最终包含两个类：互斥类和一个 `calculator` 类。下面是 `calculator` 类的声明：

```
class calculator: public query_processor{
protected:
    double Result;
    double processOperator(expression_component Operation);
public:
    calculator(string Input);
    calculator(string Input, VSet Voperands,
    VSet Voperators,
    VSet Vfunctions,
    VSet Vcharacters,
    VSet Vletters,
    VSet Vnumbers);
    calculator(void);
    double result(void);
    void processCloseParenthesis(void);
```

```

void evaluate(void);
};

```

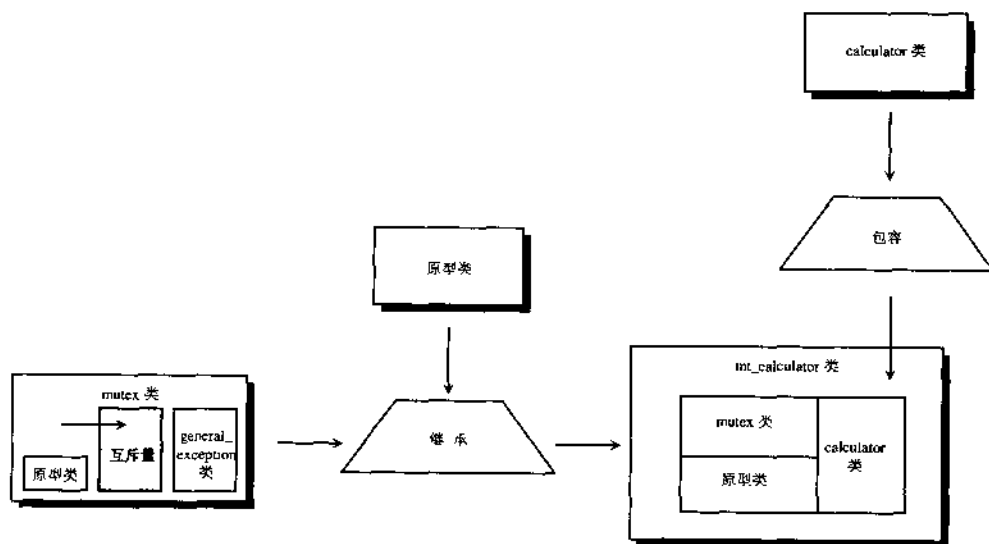


图 9-12 mt_calculator 类的类关系图

calculator 类为 mt_calculator 类的私有数据成员。进一步考察后，我们发现 mt_calculator 类是一个针对 calculator 类的接口类。类 calculator 没有提供自己的锁定和同步，因此，在多线程应用程序中使用这个类，我们为它提供一个具备锁定和同步的新接口。这是一个 C++ 类如何使用和适应新环境的典型例子。虽然 calculator 类的接口最先不是设计用在多线程环境中的，但我们可以修改接口，让它能够在多线程环境中使用。mt_calculator 类有两个主要目的：

- 为 calculator 类提供新接口。
- 结合 calculator 与互斥类。

mt_calculator 类的提供者可以为该类的用户提供线程安全服务，它实际上是通过 calculator 类来实现的。mt_calculator 类的用户同步使用 calculator 对象要做的工作减少了。mt_calculator 的用户可以编写自由不受 API 线程或锁定函数调用束缚的代码。例如，程序员可以编写如下代码：

```

#include <iostream.h>
#include "calculate.h"

void main(void)
{
    mt_calculator Calculator;
    cout<< Calculator.evaluate("98-5)*2+sin(4-2)");
}

```

如果 evaluate() 成员函数被多个线程同时调用，Calculator 的 evaluate() 成员函数的用户不必担心会发生竞争条件或死锁。mt_calculator 类的互斥量、锁定、取消锁定以及临界区对于类的用户是透明的，而且是自动发生的。这再次证明接口类对于修改现有代码用来重用是一种有

用的工具。

二、将责任从客户转移到提供者

当多线程处理面向对象架构时，转移责任给我们带来了—个重要的途径。同步对象访问以及保护对象临界区的责任应当从对象的用户身上转移到对象的提供者身上。对象的用户不用费心去规划对象具有的临界区，在使用私有属性时，这一工作可能相当困难，对象的提供者必须提供同步和保护措施。例如，有一个访问某共享版本 calculator 对象的多线程应用程序：

```
calculator Calc;

threadA()
{
    lock();
    Calc.getInputString(sin(7)+5);
    Calc.parse();
    Calc.evaluate();
    unlock();
}

threadB()
{
    lock();
    Calc.getInputString(3+2);
    Calc.parse();
    Calc.evaluate();
    unlock();
}

main()
{
    create(threadA);
    create(threadB);
}
```

因为 Calc 对象没有提供自己的锁定和取消锁定处理，所以 Calc 对象的用户承担保护 Calc 对象临界区的责任。锁定和取消锁定处理必须在类的外部由外部函数完成。在以上的伪码中，threadA()和 threadB()使用了 Calc 对象外部的锁定和取消锁定函数。虽然临界区属于 Calc 对象，但因为 Calc 不是线程安全的，它只能被 calculator 类型的对象用户从外部进行保护。对于 calculator 类的用户来说，这样做有一定难度。这样做用户必须知道 calculator 类的临界区。这一工作并不总是一项轻松的任务。类的私有部分特别难以标识和协调。

第二点，对于哪一个成员函数修改了对象内部状态，也不能轻易弄清楚。在某些情况下，用户的工作陷入一种盲目猜测之中。如果更改类的实现又将如何呢？会有新的临界区不知不觉地引入吗？最后，我们所处的境地是，一个线程正确使用互斥量，而另一个线程遗忘了该互斥量，例如：

```
calculator Calc;

threadA()
{
    lock();
    Calc.getInputString(sin(7)+5);
    Calc.parse();
    Calc.evaluate();
    unlock();
}

threadB()
{
    Calc.getInputString(3+2);
    Calc.parse();
    Calc.evaluate();
}
```

```
main()
{
    create(threadA);
    create(threadB);
}
```

请注意, `threadA()` 在 `Calc` 对象的周围放置了 `lock()` 和 `unlock()` 函数, 而 `threadB()` 没有包含这样的保护。在拥有成百上千行多线程代码的大型程序中, 是很容易犯这种错误的。这种错误通常是由于更改原有系统而产生。`threadB()` 中所犯的错误通常是由于在现有多线程系统中引入了一个新线程而产生的, 因为程序员并没有把握所有的临界区。将锁定的责任转移到 `calculator` 类的提供者身上后, `calculator` 类的用户就可以避免以上或其它竞争条件方面的缺陷。

当由类的提供者负责保护临界区后, 互斥量和条件变量在类的内部使用, 而不是在类的外部使用, 例如:

```
double mt_calculator::evaluate(string Input)
{
    double Temp;
    lock();
    Calculator.inputString(Input);
    Calculator.process();
    Temp= Calculator.result();
    unlock();
    return(Temp);
}
```

这里的 `evaluate()` 成员函数提供了自己的锁定和取消锁定处理。用户可以编写以下代码:

```
mt_calculator Calc;

threadA()                                threadB()
{                                          {
    Calc.evaluate(sin(7)+5);              Calc.evaluate(3+2);
}                                          }

main()
{
    create(threadA);
    create(threadB);
}
```

它没有包含任何锁定和取消锁定函数调用。所以, 如果我们有二个线程, 它们都使用一个安全 `Calc` 对象, 用户就免去了编写同步代码的麻烦。

三、对象访问策略

面向对象编程的主要优点之一是封装为数据提供的保护。封装用作数据用户与数据之间的中介。封装可以提供对象访问策略 (object-access policy) 以及使用原则。例如, 某矢量类可能定义下标运算符, 在使用非法下标或导致矢量返回矢量中的第一个或者最后一个对象时, 它抛出异常。这是一个由类的提供者作出的决策。C++ 类中的另一个决策例子是当发生被 0 除的情况时怎

么办。类的提供者可以返回一个错误或异常，或终止程序。这是一个由为类的提供者决定的对象访问策略。类的接口控制如何访问类的数据。请记住，对于可能应用于多线程环境中的类，我们必须包含对象访问策略。提供对多线程处理的支持以及对类的临界区的保护是类提供者的一部分责任。如果打算将类用于多线程应用程序中，则必须为类中的临界区指定访问策略。类的设计者必须提供类的相应并发模型。

四、逻辑 PRAM 模型

C++编程语言的主要优点之一是它可以用于低级和高级编程。也就是说，C++可以用来直接操纵计算机的硬件部分，例如寄存器（register）、DMA 通道（直接存储存取通道）、地址端口（address port），或其它任何与计算机相连的周边设备。C++容易与计算机汇编语言（assembly language）集成。同时，C++也可以用于实现高级概念，例如数据库管理系统（database management system）、离散事件模型（discrete event model）、黑板（blackboard）、面向对象客户机/服务器（object-oriented client/server）等等。它所具备的高级表达能力，正是我们在使用 C++构建多线程架构时最感兴趣的。

因为不同类型的并行硬件架构如此之多，操作系统支持的不同并发模型也如此之多，所以，C++类的设计者需要一种不依赖于系统的方式来实现并发性和并行性。支持并发的架构有多种。表 9-4 列出了支持并行和分布式处理的部分常见架构。

表 9-4 支持并行和分布式处理的部分常见架构

并行架构	描 述
SIMD	单指令多数据流（Single Instruction Multiple Data Streams）利用中心控制单元、多处理器以及相互连接网络进行进程到进程或进程到内存间的通信。控制单元对执行的所有处理器广播单条指令。相互连接网络允许在一个处理器上计算结果的指令与其它处理器通信。它然后可以在后续指令中用作操作数
处理器数组	同时所有数组元素上执行相同的指令。用于大型科学计算。它包括使用大量单位处理器
关联内存处理器	根据内容使用特殊比较逻辑来访问并行保存的数据
MIMD	多指令多数据流（Multiple Instruction Multiple Data Streams）利用那些使用本地数据执行独立指令流的多个处理器。此架构在计算机中与分散硬件控制异步
分布式内存	连接处理节点与处理器到处理器相互连接网络。节点通过相互连接网络消息传递共享数据。分布式内存架构例子有： <ul style="list-style-type: none"> • 环形拓扑（ring topology） • 网状拓扑（mesh topology） • 树状拓扑（tree topology） • 超立方体拓扑（hypercube topology） • 可重构拓扑（reconfigurable topology）

续表

并行架构	描 述
共享内存	<p>通过提供全局、共享内存完成处理器协调。每个处理器都可以编址共享内存。处理器要求原子同步机制来防止另一个进程完成更新它前进程访问数据。以下是用于让多个处理器访问共享内存的方法示例：</p> <ul style="list-style-type: none"> • 总线互联 • 交叉互联 • 多级互联

程序员不可能时刻清楚目标计算机将是什么物理架构。所以，当我们不知道相应的并发模型是什么时，如何来设计支持并发或多线程处理的类呢？提供系统独立 API 的当前趋势是通过软件层提供对特定硬件实现的支持。这种技术允许程序员为身边的特定硬件部分设计特定的设备驱动程序（device driver）。这个设备驱动程序称作物理设备驱动程序（physical device driver）。实现物理设备驱动程序后，就在物理设备驱动程序的顶层实现了虚拟设备驱动程序（virtual device driver）。实现虚拟设备驱动程序后，就在虚拟设备驱动程序的顶层实现了设备上上下文（device context）。图 9-13 显示了典型的软件层，它涉及何时需要与软件独立的访问。

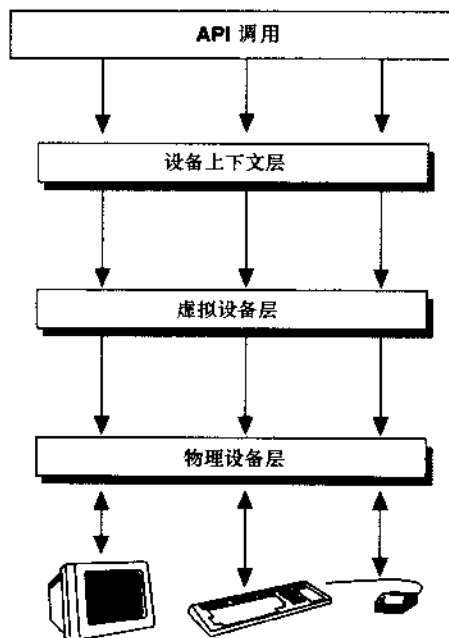


图 9-13 包含何时需要软件独立访问的典型软件层

系统独立 API 的用户只操纵设备上上下文，它是一种系统独立、高级数据结构，这个数据结构表示程序员需要操纵的特定设备。实质上，特定硬件代码在软件层被尽可能地推下，让物理设

备驱动程序位于最低层。

为了开发支持并发的面向对象架构，C++程序员可以采取相同的方式。首先，在一套低层类中封装特定系统的功能性，低层类例如线程类和互斥类，同时，在低层类的上层构建高层类。最后，最高层类的用户中只能看到一般性的并发模型。PRAM（并行随机存取机器，Parallel Random Access Machine）模型就是这样一个模型。PRAM 模型的基本组件如图 9-14 所示。

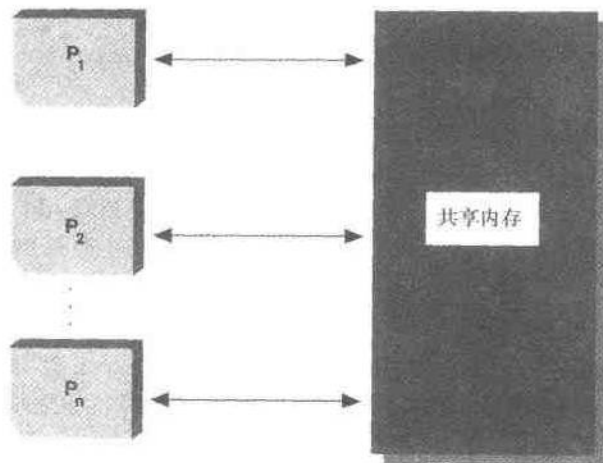


图 9-14 PRAM 模型的基本组成，一般由一套连续的处理器表示，这些处理器并行地读取或写入全局存储器

PRAM 模型是考察并行架构的简单化方式。PRAM 模型一般由一套连续的处理器 ($P_0, P_1, P_2, \dots, P_n$) 来表示，这一套处理器存取一个共享的全局存储器。所有处理器可以并行地读取或写入全局存储器。处理器也可以并行地执行不同的算术和逻辑运算 (Cormen, Leiserson, Rivest, 1995)。对于处理器执行指令的顺序没有既定的假设。只是假设所有处理器以同一速度来执行。也就是说，没有哪个处理器比其它处理器快。还假定每个处理器使用相同的定时限制，按相同的方式访问共享存储器。虽然 PRAM 模型描述硬件架构，但我们可以修改它，用它来描述软件架构。通过用线程替换处理器，用对象替换共享存储器，我们可以实现逻辑 RAM 的思想。图 9-15 显示了可以用于多线程设计的逻辑 PRAM 组件。

将逻辑 PRAM 模型应用到多线程环境中使用的类中，程序员可以设计具有高层接口的类，这个接口几乎可以被任何并行硬件或操作系统架构支持。类的数据组件看作共享、可修改的全局存储器，访问数据组件的线程看作单独、并发执行的处理器。事实上，在多处理器系统中（一个系统具有多个处理器），每个线程可以在各自的处理器上执行。在这些系统上的 PRAM 和逻辑 PRAM 是相同的。

在 PRAM 模型中操作的算法一般可分为 4 类并发访问 (Cormen, Leiserson, Rivest, 1995)：

- EREW——排它性读和排它性写；
- CREW——并发读和排它性写；
- ERCW——排它性读和并发写；
- CRCW——并发读和并发写。

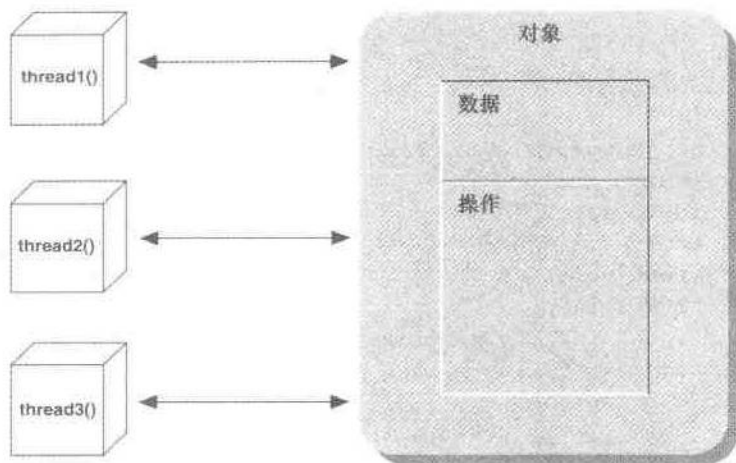


图 9-15 用于多线程设计的逻辑 PRAM，其中多个线程访问单个对象

所有 PRAM 算法都可以归入以上 4 种类型中。在逻辑 PRAM 中，排它性读某一时刻只允许一个线程访问数据组件。并发读允许多个线程同时访问对象。排它性写在某一时刻只允许一个线程修改对象。并发写允许多个线程同时修改对象。4 种并发访问类型的每一种类型都需要不同级别和不同类型的同步。EREW 需要的同步数量最少，因为不可能让两个线程同时访问一个全局对象。另一个极端是 CRCW PRAM，所有的全局对象都可能被多个线程并发写和读。显然，为了维护数据的完整性和系统操作性能，CRCW 访问将需要最多的同步和协调。在这里，我们只讨论 EREW 和 CREW 类型。

我们设计一个 EREW 或 CREW 类。典型情况下，C++ 类具有一个数据组件和一个代码组件。数据组件被写入或从中读取。代码组件完成读和写。当我们将某个类设计为 EREW 时，就意味着类的这个数据组件在某一时刻只能被一个线程访问来进行读或写。当我们把一个类设计为 CREW，就意味着类的数据组件将支持并发执行读的多个线程，但在某一时刻只能由一个线程修改此数据组件。当然，还有可能存在混合类，部分数据组件支持 EREW 访问，而部分支持 CREW 访问。CREW 访问也允许多个线程在一个线程写入数据组件时读取数据组件。如果适当分割数据组件，就可以控制这种情况，让正被写的部分数据组件不同于被写入的数据组件部分。

通过类成员函数实现 EREW 策略

通过结合同步成员函数（通常是互斥量成员函数）与宿主类的成员函数来实现 EREW 访问策略。例如，程序清单 9-2 中的 lqueue 类就是如此。

程序清单 9-2 lqueue 类是一个接口类，它结合了 deque 类、named_mutex 类和 event_mutex 类

```
#include <deque.h>
#include "ctmutex.h"

template <class T> class lqueue : virtual private named_mutex,
```



```

        virtual private event_mutex{
protected:
    deque<T> SafeQueue;
public:
    lqueue(char *MName,int Own,char *EName,int Initial,
        unsigned long Dur);
    inline void insert(T X);
    inline T remove(void);
    inline T front(void);
    inline T back(void);
    inline unsigned int empty(void);
    inline unsigned int size(void);
    inline void erase(void);
    void wait(void);
    void broadCast(void);
};

template <class T> lqueue<T>::lqueue(char *MName,int Own,
        char *EName,
        int Initial,
        unsigned long Dur) :
        named_mutex(MName,Own),
        event_mutex(EName,Initial,Dur)
{
}

template <class T> void lqueue<T>::insert(T X)
{
    lock();
    SafeQueue.push_back(X);
    unlock();
}

template <class T> T lqueue<T>::remove(void)
{
    T Temp;
    lock();
    Temp = SafeQueue.front();
    SafeQueue.pop_front();
    unlock();
    return(Temp);
}

```

程序清单 9-2 中的 `lqueue` 类是一个接口类，它修改了 STL 双端队列容器的接口。`lqueue` 类用

作一个宿主来结合 deque 类与 named_mutex 类和 event_mutex 类。这允许我们为用户提供一种在多线程环境中访问 STL 双端队列的方式。请注意，没有为 STL 迭代器提供访问方式。迭代器概念在多线程环境中难以实现。在这种情况下，为了让例子易于被读者理解，我们删除一些功能进行了简化。remove() 和 insert() 成员函数演示了 EREW 访问策略。insert() 成员函数用于给 lqueue 容器添加对象。remove() 成员函数用于从 lqueue 容器中取出对象。两个成员函数都使用了继承的 lock() 和 unlock() 成员函数。这意味着，如果多个线程访问 lqueue 类型的对象，在某一时刻只能有一个线程能写或读这个对象。这就是 EREW 的含义（排它性读，排它性写）。EREW 访问是一个对象访问策略，它决定多个线程如何才能访问或修改某个对象的数据组件。

逻辑 PRAM 有 N 个线程 $t_1, t_2, t_3, \dots, t_N$ ，它们能够访问一个全局对象 X 。记住，当我们谈到线程访问全局对象时，这个线程只是通过对象成员函数访问该对象。这意味着，是对对象成员函数强制实施 EREW 和 CREW 访问策略。对于 lqueue 类，是 insert() 和 remove() 成员函数强制实施 EREW 对象访问策略。

9.4.2 类成员函数 CREW 策略

CREW 访问策略比 EREW 的限制性少。CREW 允许并发读，而 EREW 只允许排它性读。具有静态部分数据组件的对象可以允许多个线程同时访问这些组件。这些静态数据组件有查找表、窗口句柄、设备上下文、文件描述符、基地址等等。如果共享数据不可修改，就没有竞争条件的线程。访问数据组件的成员函数只是为了读取可能声明为常量成员函数（const member function）的数据。常量指派方式保证了成员函数不会修改对象的数据。另一方面，CREW 允许发生单一、排它性写的同时进行读。这可能是一种危险的情形。如果读操作是访问写操作正更新的同一个数据组件，就导致竞争条件。类的设计者必须确保类的 CREW 访问策略能防止这种现象的发生。例如，某个类可能有几个数据组件，例如如下所示的 mcidevice 类：

```
class mcidevice{
protected:
    unsigned int DeviceId;
    char DeviceName[20];
    char Media[33];
public:
    mcidevice(void);
    mcidevice(char *DName, char *MName);
    void open(char *DName, char *MName);
    unsigned int deviceId(void);
    void media(char *X);
    ~mcidevice(void);
    void play(void);
    unsigned long capability(unsigned long Cap);
};
```

mcidevice 类表示一个媒体设备，它可以为声卡、NTSC 卡、扫描仪、CD 播放器、VCD 播放器或 VCR。一旦创建了 mcidevice 类型的对象，它就具有一个 DeviceId，DeviceId 在该对象的生存期间保持不变。这意味着，DeviceId 可以被多个线程并发读取，而不会发生任何问题。而且，

多个线程可以使用 `media()` 函数写 `Media` 组件；多个线程使用 `deviceId()` 函数读取 `DeviceId`。只是因为数据组件的本质的原因，才允许进行这种并发访问。如果类的设计者想赋予类 `CREW` 访问，他必须准备将类的数据组件分割成可能同时发生的读和写集合以及必须分离进行的读和写集合。很显然，类的设计者必须提供支持对象并发访问的对象访问策略。

9.5 增量多线程处理

下面讨论多线程处理面向对象架构的增量途径（incremental approach）。增量途径使用软件构建块和对象访问策略来实现多线程架构。首先，构建同步、任务和线程类。这些类包括互斥量、管道、队列、条件变量、事件、线程和任务类。它们将是封装操作系统支持多线程和多任务处理的接口类。这些类是低层次的。所有的特定操作系统和特定硬件代码都将在这些类中实现，或在这些类的祖先类中实现。针对使用这些类的每种操作环境，它们一般有一个对应的版本。完成这些低层构建块后，通过继承或复合将它们连接到宿主类。在 `lqueue` 类中，我们已经见到了这样的例子，`lqueue` 是一个宿主类，我们通过继承将 `named_mutex` 类和 `event_mutex` 类与 `lqueue` 连接。`lqueue` 类还通过包容（也称做复合）包含一个 `deque` 类。图 9-16 显示了 `lqueue` 类的类关系。

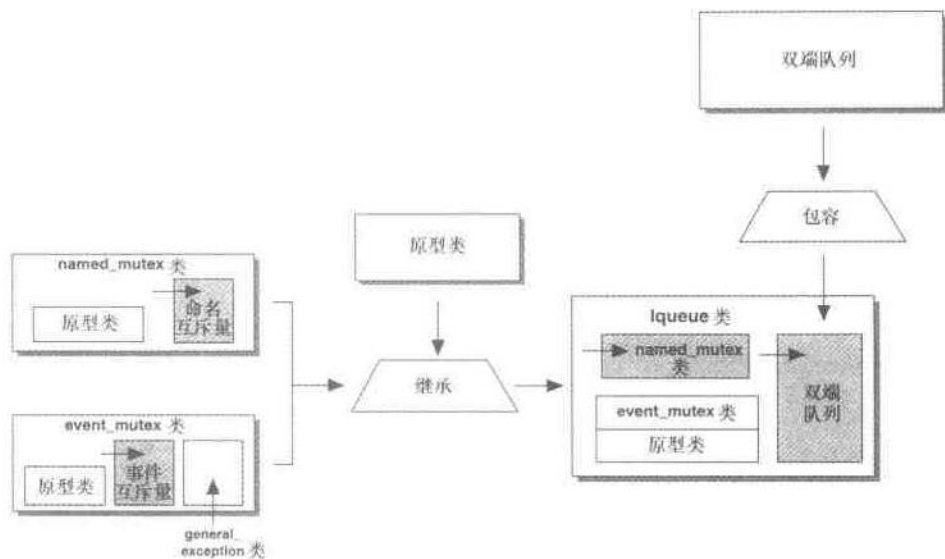


图 9-16 `lqueue` 类的类关系图，它包含一个通过包容的 `deque` 类，而且继承了 `named_mutex` 和 `event_mutex` 类

`lqueue` 类的结构是一种可以用于多线程环境的典型面向对象架构。这个基本架构包含一个互斥类和一个事件类。它是另一个不安全类的典型接口类。在这里，`lqueue` 是 STL `deque` 类的一个接口类。STL `deque` 类不具有对多线程环境的任何内置支持，所以，我们通过 `lqueue` 类修改它的接口。宿主类与适当的同步类结合后，类设计者决定该对象支持何种类型的 `PRAM` 访问策略。这

种策略可以是 EREW、CREW、ERCW 或 CRCW。

也可以混用这些策略。不过,实现混用需要一些技巧。一旦设计者为类选定了并发访问策略,必须相应地设计该类的成员函数,使得类用于多线程环境中时,强制实施并发访问策略。这要求与类连接的同步类应当用于正确保护临界区。它还要求设计者将类的数据组件分割成可以并发修改和访问的部分以及不能并发修改和访问的部分。锁定和取消锁定通过所需的成员函数来完成。用信号通知和广播由所需的成员函数完成。类设计者在可使用常量的地方使用常量。通过类成员函数实现类的访问策略后,该类就可以用作一个任务类、线程类或应用框架中的组件了。接着让任务类、线程类或应用框架适应于支持多线程处理的范例(如客户机/服务器、事件驱动和黑板)。我们称这一过程为增量多线程处理(incremental multithreading),因为它是分阶段有系统地来实现。在设计方,它从选定适宜于多线程处理的架构开始。一旦选定了架构,然后设计应用框架。应用框架设计完毕,再设计支持应用框架的类库和容器类。类库、容器类和域类都基于同步类而构建。图 9-17 显示了多线程面向对象架构的结构。

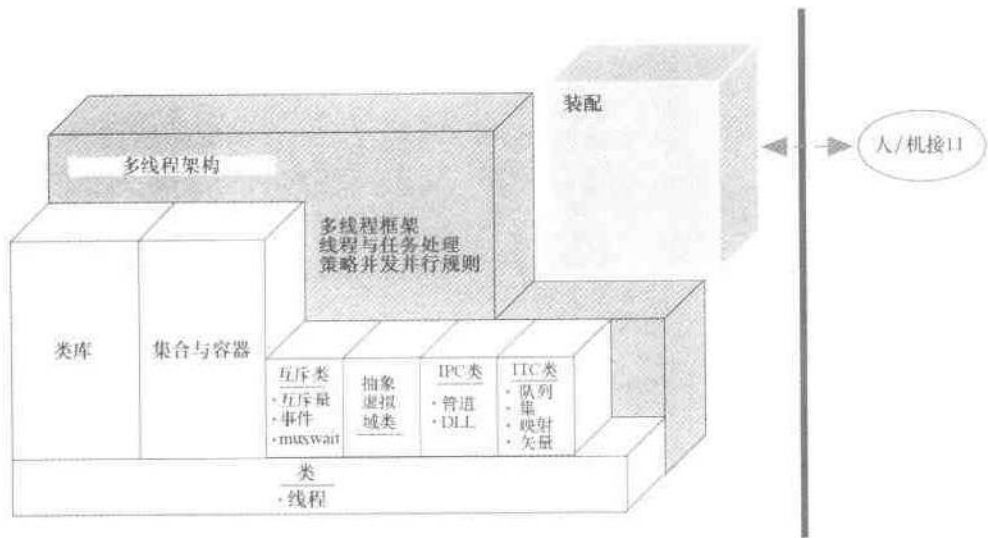


图 9-17 多线程面向对象架构的结构

在实现方,从最低层的类开始,向最高层类推进。它是一种构建块途径来多线程处理 C++ 组件。它不是试图强迫应用于多个线程,而是选定一个自然而然适宜于多线程处理的架构。然后,我们用一些提供了同步和协调的组件来填充这些架构。构建多线程面向对象架构的一般原则分为两类:

设计方面:

- 选定一种容易多线程化的架构。
- 使用一种应用框架来实现这个架构。
- 使用容器类和类库来实现应用框架。
- 对适当的同步对象构建容器类、类库和域类。

实现方面：

- 创建同步类。
- 结合同步类与宿主类。
- 选定对象访问策略（CREW、EWEW 等等）。
- 使用成员函数来强制实施实现策略。

现在，我们知道了如何实现 C++ 组件的增量多线程处理，以后我们将探讨用哪些类型的 C++ 类来构建多线程应用。我们已经探讨了多线程应用中接口类的许多用途。下一章，我们将探讨其它类型的 C++ 类，以及哪些类在构建多线程架构时特别有用。

类层次和线程处理 C++组件

它们甚至可能当场得到进化。于是形成一个通常被看作极其枯燥无味的生物分支，这一分支对于那些对关系本质感兴趣的人具有极大的价值。

The Illustrated Encyclopaedia of Dinosaurs

类层次（class hierarchy）是一些通过继承（inheritance）相关联的类的集合。类层次表示一个事物、地点、事物或思想的逻辑家族，也可以包含其它类作为数据成员。一般情况下，类层次构建于多种 C++ 类类型。类层次中的一些常见类类型有：

- 抽象基类；
- 具体类；
- 节点类；
- 接口类；
- 容器和集合类；
- 应用框架类；
- 域类。

10.1 抽象基类

抽象基类（abstract base class）为所有后代提供蓝图。它作为从它派生的所有类的一套原则和接口策略。事实上，抽象类用作类的规范。用户不能声明抽象类的实例。为了使用抽象类，用户必须首先从基类派生出一个新类，并为抽象类中声明的所有纯虚函数提供实际的定义。在程序员利用抽象基类前，他（她）必须声明一个基类的后代。缺省情况下，抽象类必须至少有一个有用的后代。这一点与具体类形成鲜明的对照。具体类不支持派生后代，抽象基类要求后代为可用类。抽象类在 C++ 语言结构中的身份是纯虚函数。对于一个抽象类，它必须至少有一个纯虚函数，其

形式如下：

```
virtual return type function()=0;
```

在多线程应用程序中，抽象基类可以用于定义充当同步化派生类原则的关系。例如：

```
class myclass{
protected:
    int A;
    int B;
public:
    virtual void lock(void)=0;
    virtual void unlock(void)=0;
    int updateA(void);
    int updateB(void);
};

int myclass::updateA( )
{
    lock( );
    //通过 A 完成一定任务
    unlock( );
    return 1;
}

int myclass::updateB( )
{
    lock( );
    //通过 B 完成一定任务
    unlock( );
    return 1;
}
```

lock()和 unlock()函数为纯抽象虚函数。它们在 myclass 中没有定义。为了让程序员使用 myclass, 必须从 myclass 中派生另一个类。派生类必须提供成员函数 lock()和 unlock()的定义。lock()和 unlock()在 updateA()和 updateB()成员函数中的使用方式为 myclass 的用户提供了将用 lock()和 unlock()来干什么的指导原则。纯抽象虚函数作为程序员如何处理特定代码段的提示。请注意，在派生类使用 updateA()和 updateB()的任何时候，访问 A 和 B 都要带上 lock()和 unlock()成员函数。如果正确定义了 lock()和 unlock()函数，在多线程应用程序中对数据成员 A 和 B 的所有访问都将受到保护。所以，要使用 myclass, 我们必须创建一个 myclass 类的后代类。myclass 和 derived_class 这两个类组成了一个简单的类层次：

```
class derived_class:public myclass{
protected:
    pthread_mutex_t Mutex;
public:
    derived_class(void);
    void lock(void);
}
```

```
void unlock(void);
};
```

如果 lock() 和 unlock() 函数使用一些适当的锁定功能来实现, 例如:

```
void derived_class::lock(void)
{
    pthread_mutex_lock(&Mutex);
}

void derived_class::unlock(void)
{
    pthread_mutex_unlock(&Mutex);
}
```

则调用抽象基类的 updateA() 和 updateB() 成员函数将最终调用 derived_class 的 lock() 和 unlock() 成员函数。通过 C++ 的运行时多态功能可以实现, 而且证明了在设计用于多线程环境的类时抽象基类的有用性。虽然纯抽象虚函数可以用于强制实施接口策略, 但它不是百分之百安全。并不能阻止派生类的设计者定义 lock() 和 unlock() 成员函数来完成锁定和取消锁定之外的其它事。设计者可以将它们实现为哑元函数, 例如:

```
void derived_class::lock(void)
{
}

void derived_class::unlock(void)
{
}
```

另一方面, 设计者可能对 lock() 和 unlock() 作出其它特殊的解释, 这些解释可能与本来意图完全相反, 例如:

```
void derived_class::lock(void)
{
    // 抢占中断表
}

void derived_class::unlock(void)
{
    // 释放中断表
}
```

在这种情况下, 就失去了在基类中所声明的使用 lock() 和 unlock() 成员函数的意图, 因此要记住, 纯抽象虚函数仅作用如何构建派生类的接口原则与蓝图。纯抽象虚函数不能强迫派生类实现执行特定任务的函数。不过, 如果基类诠释得当, 就可以用纯抽象虚函数与派生类具有的正确架构进行沟通。抽象基类通过指定必需的函数及其关系, 它可用作多线程架构的基础。

10.2 具体类——理想终结者

C++中的许多类设计作为基或祖先类的候选者，具体类作为终结类来设计和实现。一般来说，具体类为独立的类，它表示祖先-后代世系中的结束。在一个类层次中，具体类或者是结束层次的终结类，或者是类层次中另一个类的数据成员。让我们看看一个包含两种类型类的类层次。第一个类 `query_processor` 是一个为查询或命令解析器建模的类。这个类表示一个字符串样式的命令或查询。`query_processor` 类是一个抽象基类。

```
class query_processor{
protected:
    set<char,less<char>>>ValidOperands;
    set<char,less<char>>>ValidOperators;
    set< string,less< string >>>ValidFunctions;
    set<char,less<char>>>ValidCharacters;
    set<char,less<char>>>ValidLetters;
    set<char,less<char>>>ValidNumbers;
    string InputString;
    list<expression_component>ExpressionTokens;
    stack<vector<expression_component>>>OperandStack;
    stack<vector<expression_component>>>OperatorStack;
    int ValidExpression;
    char FunctionReult[10];
public:
    query_processor(string Input);
    query_processor(string Input,VSet VOperands,
                    VSet VOperators,
                    VSet VFunctions,
                    VSet VCharacters,
                    VSet VLetters,
                    VSet VNumbers);
    query_processor(void);
    int checkParenthesis(void);
    int checkOperators(void);
    int checkDecimals(void);
    int checkCharacters(void);
    virtual void evaluate(void)=0;
    int parse(void);
    string inputString(void);
    void inprtString(string Input);
    int validExpression(void);
    double processNumber(int &Count);
    char *processFunction(int &Count);
    list<expression_component>expressionToken(void);
    void process(void);
```

```

        int validate(void);
    };
    它包含纯抽象虚成员函数 evaluate():
    virtual void evaluate(void)=0;
    query_processor 类将要用作某类层次的基础。我们下面要考察的第二个类是 calculator 类:
    class calculator:private query_processor{
    private:
        double Result;
        double processOperator(expression_component Operation);
    public:
        calculator(String Input);
        calculator(String Input,VSet VOperands,
            VSet VOperators,
            VSet VFunctions,
            VSet VCharacters,
            VSet VLerrers,
            VSet VNumbers);
        calculator (void);
        double result (void);
        void processCloseParenthesis(void);
        void evaluate(void);
    };

```

calculator 类继承了 query_processor 类。calculator 类可以接受如下所示字符串:

```

4+(6*3)
12.5/sin(0.5)+(9.3-2)
7/22+(4/3+5)

```

calculator 类是一个具体类。它的目的不是作为后代类。请注意, 它私有地继承了 query_processor, 而且两个重要的成员也声明为私有。如果继承这个类, 用户不得不做许多额外的工作来让这个类完成原始设计之外的其它事情。图 10-1 显示 calculator 类的类关系图。

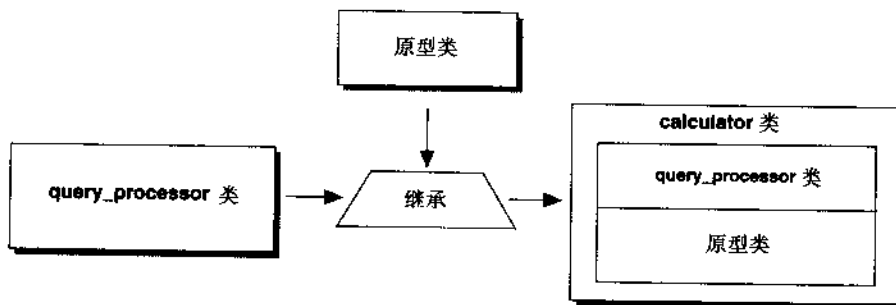


图 10-1 calculator 类的类关系图

类层次以 query_processor 类开始, 以 calculator 类终止。“一般情况下, 具体类型不适合于相关类的一般系统。每个具体类型不必参考其它类就可以独立理解”(Stroustrup, 1991)。虽然具体

类可以构建于其它类，但具体类作为进一步继承的终点。设计 `calculator` 类不是作为一般性对象，而是用作一个计算器。`query_processor` 类的目的是具有一般性，它要用作一个基类，将 `calculator` 类用作一个基类就难以实现。一旦实现了具体类，在以后的后代类中就不会再特殊化。具体类可以在实现中进行优化，因为不必担心派生类中的功能性（Stroustrup, 1991, 431~435）。这使得 `calculator` 参与多线程处理成为可能。

在类层次中使用多线程处理的一个危险是，派生类可能覆盖成员函数并毁坏在类层次中实现的保护策略。派生类通常更改成员函数的实现，而且有时更改在基类中声明的函数的意义。这可能会导致问题的发生。例如，如果 `myclass` 的 `lock()` 和 `unlock()` 成员函数如下所示：

```
class myclass{
protected:
    int A;
    int B;
public:
    virtual void lock(void)=0;
    virtual void unlock(void)=0;
    int updateA(void);
    int updateB(void);
};
```

成员函数在派生类中被重新定义，已经不再用于互斥量锁定和取消锁定，于是，将来多线程应用程序中 `updateA()` 和 `updateB()` 的执行可能导致数据竞争。另一方面，如果我们基于线程安全基础或多线程类层次为用户提供一个具体类，那么我们就无需担心用户在具体类中对成员函数意义的改变。具体类没有虚函数。它不打算用来被继承。它只是按现在的样子进行直接使用。我们可以把具体类用作包含安全类和多线程类的类层次的终点。按这种方式，具体类的用户在线程环境中就拥有了多线程处理或保护的优势，而无需担心破坏类层次的语义。

10.2.1 多线程层次中的节点类

也许 C++ 编程中使用的最强大类类型就是节点类（node class），它提供继承和多态的基础。与抽象基类一样，节点类设计用于被继承。与抽象基类不一样的是，节点类不包含纯虚函数。节点类可以立即使用。不过，节点类的设计还同时着眼于未来。它设计具有可重用性，同时提供可以在派生类中覆盖的虚成员函数。它提供可以被派生类继承的受保护数据成员和成员函数。它使用基类的指针（pointer）和引用（reference）使得在后代类中可以操纵成员函数。用户可以通过多态和继承特殊化节点类。节点类既可以是一个基类，也可以是一个派生类。节点类提供了类层次的基础。

在多线程层次中维护节点类需要一定的技巧。因为节点类使用了基类的指针、引用、虚函数和运行时多态，所以可能会误用节点类，而使多线程或线程安全类层次失效。当类层次定义明确、能准确理解时，节点类应当只用于多线程或安全的类层次中。C++ 程序员修改这个层次时，他应该完整理解了运行时多态（runtime polymorphism）、深层复制构造函数（deep copy constructor）、异常处理（exception handling）以及签名分辨率（signature resolution）。在多线程环境中，节点类可能会增添混乱。除非程序员是专家，否则务必小心！

10.2.2 线程与容器和集合类

另一个重要的 C++ 类类型是容器类 (container class)。与节点类和接口类一样, 容器类没有特殊关键字或特殊 C++ 结构的支持。容器类是常规 C++ 类的一个特别应用。集合和容器为充当其它对象组一般性保存器的对象。集合和容器是一般性目的的分组结构。一些常用的容器类如下:

- 数组和矢量类 (Array and Vector Class);
- 列表类 (List Class);
- 双端队列类 (Deque Class);
- 队列和优先权队列类 (Queue and Priority Queue Class)。

开发者或设计者可以使用集合和容器来操纵异类或同类对象。开发者可以声明一个窗口队列, 或声明一个设备驱动程序的矢量。集合和容器用于管理一组对象, 其方式与传统数组管理传统数据类型 (如整数或字符) 的方式一样。使用 C++ 类和结构 (struct) 这一结构来定义集合和容器。这意味着继承、多态和封装的优势都可以应用于集合和容器中。C++ 标准库现在包括 STL (Standard Template Library, 标准模板库), 它是用作具体类的容器类的一个库。

一、ITC 的集合和容器类

与大部分 C++ 组件类型一样, 在多线程程序中, 集合和容器类与线程交互的主要途径有两种:

- 线程间通信;
- 多线程服务器。

在多线程程序中集合和容器的最基本用途是作为 ITC (线程间通信, interthread communication) 的一种机制。两个进程间通信时, 程序员传统上使用管道、命名管道、动态数据交换、命令行参数、共享内存或文件。这些结构作为多个执行进程间的外部链接, 让这些进程相互间通信与传递数据或命令。每种结构都受操作系统的支持, 而且通常通过调用操作系统 API 来管理。这些对操作 API 的调用包括创建、同步化、阻塞、取消阻塞、缓冲、关闭以及销毁与这些结构相关联内存的开销。线程相对于进程的一个主要优点是, 线程为了通信不要求这些外部数据结构。两个线程可以通过一个全局变量、一个全局数组、一个全局结构, 或通过传统参数传递来通信。在面向对象程序中, 两个线程可以通过全局容器和集合类来通信。这些集合和容器类通常具有如下形式:

- 面向对象列表;
- 面向对象矢量;
- 面向对象集 (set);
- 面向对象多集 (multiset);
- 面向对象图表;
- 面向对象双端队列;
- 面向对象队列;
- 面向对象映射;
- 面向对象多重映射。

与一般情况下将数据结构用作多个进程间通信机制相比,使用面向对象组件进行线程间通信具有巨大的优点。进程间的数据通信通常位于线性数据结构中,例如队列和管道中。这些数据结构没有提供图表、集和映射所提供的表达力。当在进程间传递数据时,程序员被迫将通信看作多个进程间的一种简单线性数据或消息交换。因为同一个进程中的多个执行线程可以访问同一个结构,所以,程序员可以在线程间创建强大的通信,这是并发执行的进程不能轻易创建的。

搜索引擎设计和实现是今天最热门的编程领域之一。这是因为互联网上可以找到浩瀚的信息,一般的 CD-ROM 上也能保存海量的信息。互联网上的大量信息以各种各样的 ASCII 文件、HTML 文件、post script 文件、多媒体文件、全息文件以及数据库方式存在,使得需要快速获取信息的用户面临许多挑战。为了战胜这些挑战,开发商不断推出多种不同的搜索技术,例如 Web 浏览器、gopher、智能代理以及搜索向导。这些搜索技术由不同类型的搜索引擎驱动。其中实现搜索引擎的一种最有效技术是将搜索分成一系列并发执行的线程,每一个线程可以访问所需要的搜索组件。线程可以将集合和容器对象用作线程间通信的渠道。

如果我们想搜索互联网,查找包含多线程编程、并发编程、并行算法以及分布式编程研究信息的文档,但我们不希望搜索返回有关并行硬件设计、超级计算机、手术研究、新闻阅读者线索、新缝纫机穿线技术等方面的信息,此时该如何办呢?而且,如果我们希望搜索能返回直接与多线程编程、并发编程、并行算法以及分布式编程相关的信息,又该怎么办呢?我们可以设计一个多线程搜索引擎来“狂轰乱炸”式地完成这一巨大的任务。我们这里所说的“搜索引擎”泛指为执行单一模式匹配而实现的模块或模块集合。我们可以将搜索分成若干不同的任务。一个任务可以是读取互联网上所有文件的索引。(对于浩瀚的信息,谈何容易!)另一个可以是为预备匹配预搜索这些文件的索引。我们然后可以建立其它多个任务进行详细的预备性匹配。我承认这是一种暴力式的途径,不过,执行这些任务的搜索引擎最终将会完成任务。在这里通过这个例子来说明如何将集合和容器类用作多个使用执行某项任务的线程间的通信点。

我们可以指派线程 A 来读取主索引。当线程 A 读取主索引时,我们可以指派线程 B 开始从预备匹配的主索引过滤每个索引。保存预备匹配的文件名时,我们可以使用线程 C 来检查我们所需要的项,让线程 D 检查文件中我们不想要的项。一旦全面启动,这些线程将并发执行。这些线程可以共享集合和容器对象进行线程间通信。集合对象包含面向对象集(set)。容器对象包含面向对象队列(queue)。如果我们使用进程间通信方法和单独的进程,这将更为困难。使用线程和 ITC 可以让我们的工作更容易。

我们可以有 4 个集(set)。第一个集由表示当前被分析文件的单词集合组成。第二个集由表示我们感兴趣主题的单词集组成。第三个集由我们不感兴趣的单词集组成。第四个集由表示我们感兴趣的单词集的相关项组成。当线程 A 从主索引中读取文件名时,它将这些文件名及其位置放进面向对象队列中。当线程 A 将文件放进面向对象队列中时,线程 B 删除并处理这些文件名来访问它们的单个索引。线程 B 查找预备匹配时,它将匹配放到另一个面向对象队列中。线程 B 还将当前位于队列前端的文件放进集 A。一旦集 A 中存在一个文件,线程 D 和线程 C 使用基本集运算来进行相应的文件分析。基本集运算包括:

- 交集;
- 集关系;

器和集合类以适应于多线程环境的简单途径。同时还要注意, `mt_set` 类有两个私有成员, `set` 组件和 `mutex` 组件。声明这些私有成员是因为我们不想让它们用作进一步继承的基类。每一个成员函数提供了多线程环境中需要的锁定处理。下面是对 `mt_set` 类的 `set` 组件执行基本集运算的成员函数的定义:

```
template<class T> set<T,less<T>>
mt_set<T>::intersection(set<T,less<T>>>X)
{
    set<T,less<T>> Temp;
    less<T> Order;
    lock();
    set_intersection(S.begin(),S.end(),X.begin(),X.end(),
                     inserter(Temp,Temp.begin()),Order);
    unlock();
    return(Temp);
}

template<class T> set<T,less<T>>
mt_set<T>::setUnion(set<T,less<T>>>X)
{
    less<T> Order;
    set<T,less<T>> Temp;
    lock();
    set_union(s.begin(),S.end(),X.begin(),X.end(),
              inserter(Temp,Temp.begin()),Order);
    unlock();
    return(Temp);
}

template<class T> int mt_set<T>::membership(set<T,less<T>>>X)
{
    return(includes(S.begin(),S.end(),X.begin(),X.end()));
}

template<class T> set<T,less<T>>
mt_set<T>::difference(set<T,less<T>>>X)
{
    set<T,less<T>> Temp;
    less<T> Order;
    lock();
    set_difference(S.begin(),S.end(),X.begin(),X.end(),
                   inserter(Temp,Temp.begin()),Order);
    unlock();
    return(Temp);
}
```

因为 `mt_set` 类的每个成员函数自动进行它自己的锁定处理,所以比起常规 STL 集类来,它更

容易在多线程环境中使用。每个这样的成员函数都使用一个 STL 算法来执行基本集运算。

多线程搜索引擎的另一个容器组件为面向对象队列。下面是线程安全 lqueue 类的声明:

```
template<class T> class lqueue:virtual private named_mutex,
                        virtual private event_mutex{
protected:
    deque<T> SafeQueue;
public:
    lqueue(char *MName,int Own,char *EName,int Initial,
            unsigned long Dur);
    inline void insert(T X);
    inline T remove(void);
    inline T front(void);
    inline T back(void);
    inline unsigned int empty(void);
    inline unsigned int size(void);
    inline void erase(void);
    void wait(void);
    void broadCast(void);
};
```

与 mt_set 类一样, lqueue 类也是一个接口类, 它适配其中一个 STL 容器类的接口。下面是被修改的双端队列容器。就像在 mt_set 类中一样, lqueue 类的成员函数提供了自己的锁定处理, 例如:

```
template <class T> void lqueue<T>::insert(T X)
{
    lock();
    SafeQueue.push_back(X);
    unlock();
}

template <class T> T lqueue<T>::remove(void)
{
    T Temp;
    lock();
    Temp=SafeQueue.front();
    SafeQueue.pop_front();
    Unlock();
    Return(Temp);
}
```

lqueue 类的 insert()和 remove()函数修改 SafeQueue 的状态, 所以, 每个成员函数都执行一个锁定。如果另一个线程试图从这个队列插入或删除东西, 该线程就会阻塞。我们去掉了 lqueue 类的迭代器功能, 因为在多线程环境中; 使用外部迭代器具有固有的难度。mt_set 类和 lqueue 类演示了一个简单架构, 它可用于被多个线程访问的对象。图 10-3 显示了 mt_set 类和 lqueue 类的类关系。

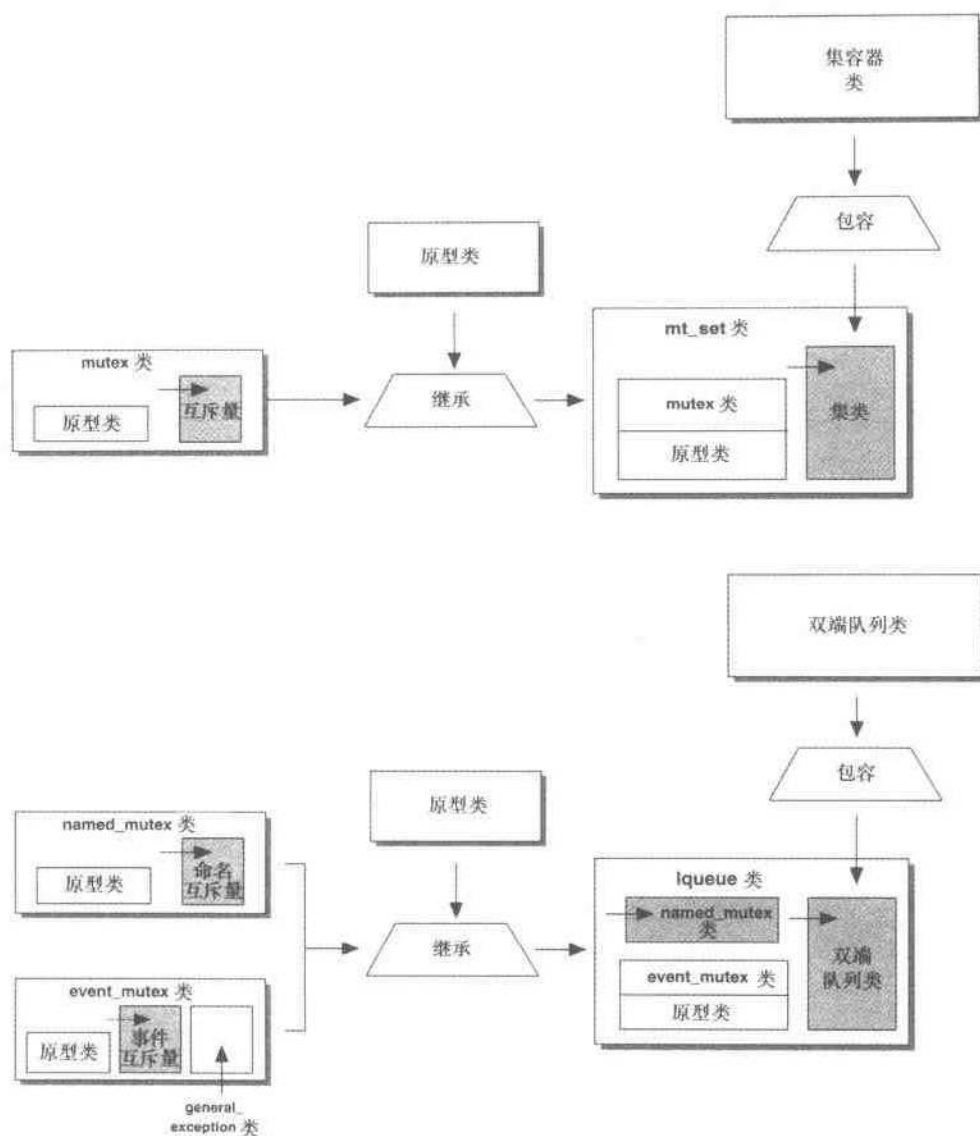


图 10-3 mt_set 类和 lqueue 类的类关系图

请注意，两个类都具有相同的结构。两个类都是修改 STL 容器类的接口类。两个类都私有地继承了用于保护类内部临界区的互斥量。多线程搜索引擎的最后一个基本组件是线程对象。面向对象架构中的每个线程都由一个对象表示。典型情况下，对象是一个封装了对应操作系统环境中 API 函数的接口类的实现。例如，`ct_thread` 是封装了一些 POSIX `pthread` API 的接口类：

```
class ct_thread{
```

```

private:
    pthread_t ThreadId;
    pthread_attr_t *Attr;
public:
    ct_thread(void);
    ct_thread(pthread_attr_t *Attribute);
    ~ct_thread(void);
    void begin(FunctionPtr PFN,void *X);
    void wait(void);
    pthread_t threadHandle(void);
    pthread_t threadId(void);
};

```

这个类用于实例化表示多线程面向对象架构中线程的对象。`ct_thread` 类是一个入门级别的线程，它包含最少的线程表示内容。在完善它之前，我必须给它添加更多的数据成员和成员函数。这个线程接口的主要优点是它隐藏了对应操作系统环境的实现细节。通过 `ct_thread` 这样的类，程序员可以设计在不同操作系统环境中可移植的程序。`ct_thread` 类的定义只是提供了 POSIX 线程 API 的封装器：

```

ct_thread::ct_thread(void)
{
    Attr=NULL;
}

ct_thread::ct_thread(pthread_attr_t *Attribute)
{
    Attr=Attribute;
}

void ct_thread::begin(FunctionPtr PFN,void *X)
{
    pthread_create(&ThreadId,Attr,PFN,X);
}

void ct_thread::wait(void)
{
    pthread_join(ThreadId,NULL);
}

pthread_t ct_thread::threadHandle(void)
{
    return(ThreadId);
}

pthread_t ct_thread::threadId(void)
{
    return(pthread_self());
}

```

}

`ct_thread` 类与 `mt_set` 和 `lqueue` 类一起表示实现多线程搜索引擎所需要的基本面向对象组件。`mt_set` 和 `lqueue` 类用于线程间通信，`ct_thread` 类用于实现查找和分析满足搜索标准的文件所涉及的多个线程。多线程搜索引擎是一个线程化应用框架的例子。我们将在后面对这个多线程应用程序作更多的讲解。

二、作为多线程服务器的容器类

如果在多线程应用程序中，容器类最常用的地方是线程间通信的话，容器类在多线程应用程序中的第二个常用地方是作为小型多线程服务器（`minimultithreaded server`）。当容器或集合类用作线程间通信的工具时，线程位于为外部。当容器或集合类用作一个服务器时，类本身将包含多个线程。我们已经看到多个线程是如何使用类的。我们已经探讨了如何使用互斥量来为类的内部临界区提供保护。

现在我们将讨论一种可用于将类分解成多个线程的简单技术。我们将使用熟悉的接口类概念来修改 `mt_set` 的接口, 得到多线程集服务器 (multithreaded set server) 的概念。新类名叫 `set_server`。图 10-4 显示了 `set_server` 类的类关系。

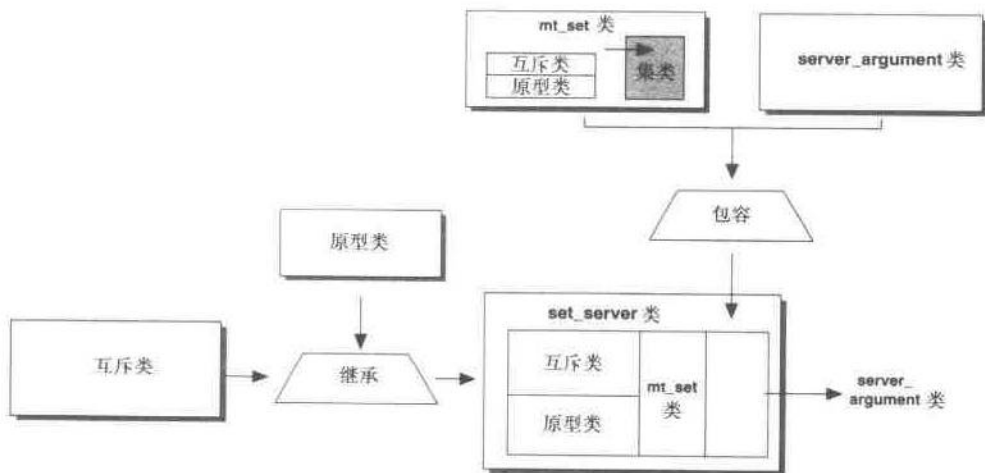


图 10-4 set_server 类的类关系图

`set_server` 类遵循用于多线程环境的类的入门级别设计。首先，它私有继承互斥类。这个互斥量用于保护 `set_server` 类中的任何临界区。其次，它使用包容关系来包含 `mt_set` 类。`set_server` 类用于提供多线程服务。多线程服务由 `mt_set` 组件来实现。这意味着 `set_server` 类创建的每个线程最终将使用 `mt_set` 类的功能来完成它的工作。如何实现呢？`set_server` 类的用户请求一个服务。`set_server` 类然后创建一个线程来执行这个服务。这个线程接着将请求传递给 `mt_set` 类。`mt_set` 完成这个请求，并返回结果。建立具有多线程架构的类的途径有多种。下面我们着重讨论建立包含线程的类的最简单方法。

多线程类的简单架构

为了设计多线程类，我们至少需要 6 个基本组件：

- 宿主类 (host class);
- 线程类 (thread class);
- 互斥和事件类 (mutex and event class);
- 友元成员函数 (friend member function);
- 域类 (domain class);
- 强制转换基本元素 (casting primitive)。

这些组件是多线程架构的基石。图 10-5 显示了这个架构的基本结构，以及这 6 个组件如何结合在一起。

宿主类是用户与之交互的类。有两种常用的多线程宿主类。第一种类型是多线程应用程序框架。第二种类型是多线程服务器。我们这里着重讨论多线程服务器。多线程容器或集合服务器的一个典型例子是面向对象数据库管理系统 (OODBS, object-oriented database management system)。宿主类被分解成两个或更多的线程。每个线程执行宿主类的一个友元函数。当执行友元函数时，宿主类将 `this` 指针作为一个参数传递给该线程。`this` 指针在友元函数中被强制转换成指向适当类型的类。`this` 指针形成了宿主类与位于另一个线程中的域类间的线程间通信。每个友元函数将创建一个或多个域对象 (domain object) 来完成线程的请求。多线程 `set_server` 的声明如下：

```
template<class T>struct server_argument{
    set<T,less<T>> A;
    set<T,less<T>> B;
    set<T,less<T>> Result;
};

template<class T> class set_server:virtual private mutex{
protected:
    server_argument<T> Argument;
public:
    set_server(void);
    friend void *intersection(void *X);
    friend void *setUnion(void *X);
    friend void *difference(void *X);
    friend void *membership(void *X);
    set<T,less<T>> intersect(set<T,less<T>> X,
        set<T,less<T>> Y);
    set<T,less<T>> setUnion(set<T,less<T>> X,
        set<T,less<T>> Y);
    set<T,less<T>> difference(set<T,less<T>> X,
        set<T,less<T>> Y);
    int membership(set<T,less<T>> X,set<T,less<T>> Y);
};
```

`set_server` 类的另一个重要组件是 `Argument` 数据成员，它用于保存多个线程间以及主机与客

户间的输入或输出参数。宿主类通常从客户（用户）类接收输入参数。输入参数保存在一些属于宿主类组件的结构中。在这里，我们声明一个包含 3 个集对象（set object）的模板类。这个类为 `server_argument` 类型。第一个集 A 保存有构建域类所需要的信息，这个域类由友元函数创建。第二个集 B 包含参数、消息或来自客户的请求。第三个集 C 包含对集 B 中信息进行操作或处理的结果。宿主类或服务器类与客户具有双向通信（two-way communication）通道。服务器类的成员函数用于实现这种通信。`set_server` 类与客户之间的关系如图 10-6 所示。

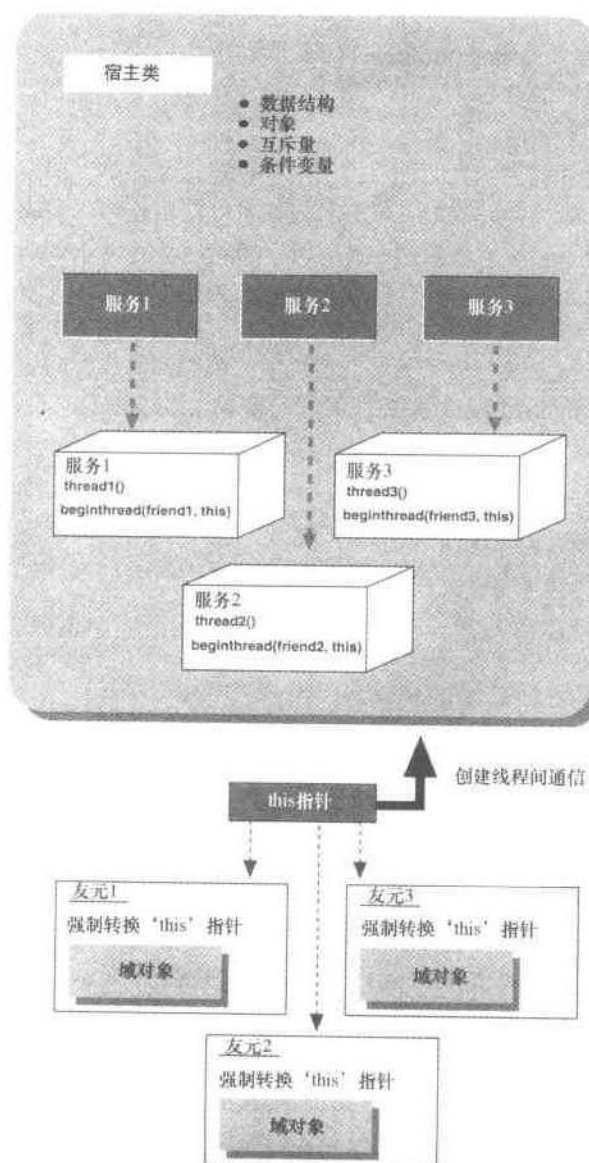


图 10-5 宿主类架构的基本结构，及其组件如何结合在一起

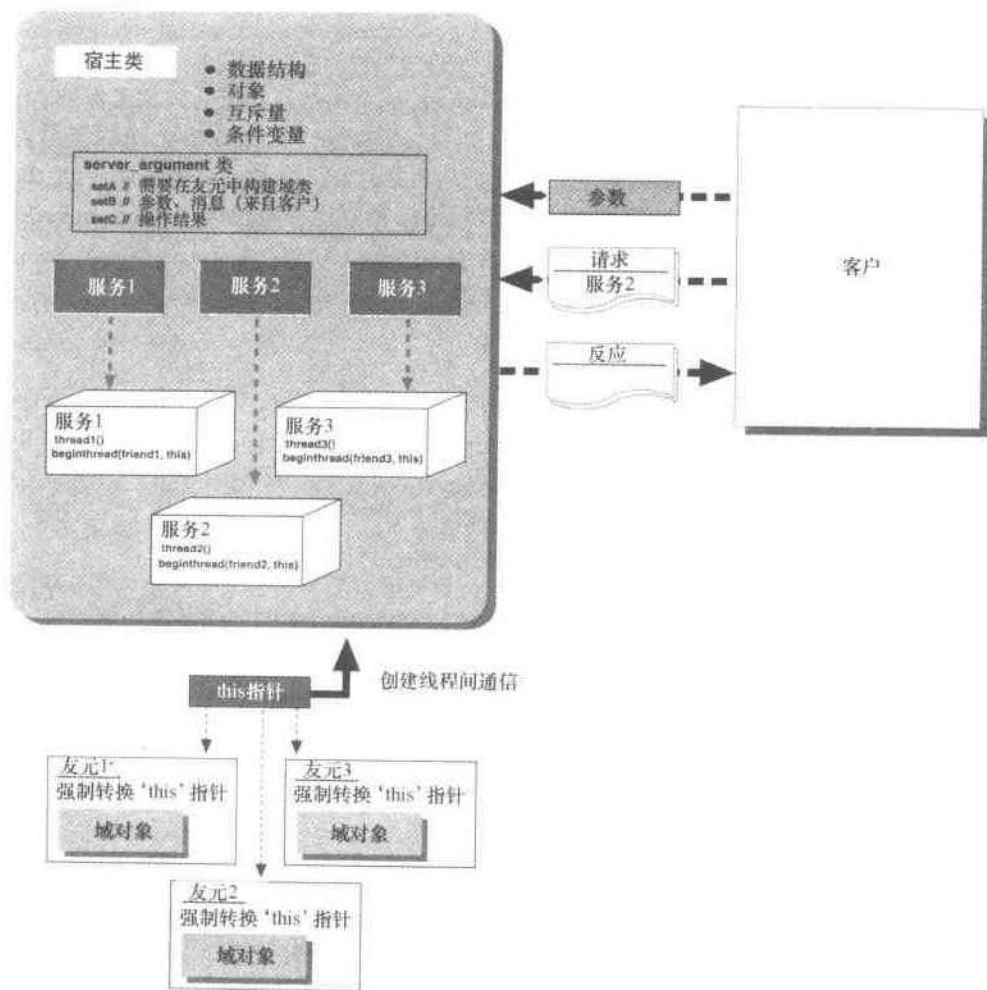


图 10-6 set_server 类与客户间的关系

每个成员函数表示一个客户的请求，所以每个成员函数创建一个执行某友元函数的线程。让我们详细看看它的实现代码：

```
template <class T> set<T, less<T>>
set_server<T>::setUnion(set<T, less<T>>X, set<T, less<T>>Y)
{
    lock();
    ct_thread Thread;
    Argument.A=X;
    Argument.B=Y;
    Thread.begin(::setUnion, this);
}
```

```

    Thread.wait();
    unlock();
    return{Argument.Result};
}

```

客户给 `set_server` 对象发送 `setUnion()` 请求。`setUnion()` 成员函数自动进行锁定来保护 `Argument` 数据成员。这个数据成员可能是潜在的临界区（如果 `set_server` 对象被多个线程使用的话）。`setUnion()` 成员函数然后创建 `ct_thread` 对象，同时通过所执行友元函数的名字来调用 `ct_thread` 对象的 `begin()` 成员函数。在这里，友元函数就是 `setUnion()`。通过运算符重载（operator overloading）可以让我们使用友元函数和类的成员函数的相同名字。这种命名约定将友元函数与类的成员函数关联起来，因此简化了过程。作用域运算符（::）是正确标识友元函数所必需的，所以我们按如下方式传递友元函数：

```
:: setUnion
```

按这种方式传递友元函数很重要，因为大部分线程创建 API 带有一个指向函数的指针，它用于调用线程即将执行的第一个函数。也就是说，函数指针成为了线程的入口点（entry point）。按这种方式，C++ 程序中的 `main()` 用作执行的主要起始点，在创建线程 API 中指定的函数用作主要起始点。例如，POSIX `pthread` 库的创建线程函数的语法为：

```
int pthread_create(pthread_t *Tid, pthread_attr_t* A,
    void*(PFN*)(void*), void *Arg);
```

其中的第一个参数为线程句柄（thread handle），第二个参数包含所创建线程的属性（attribute），第三个参数为创建线程后，线程开始执行的函数的指针。这个函数为线程的入口点。在本书中，多线程服务器和多线程应用程序框架将给线程创建服务传递友元函数指针。在 `set_server` 类中，`ct_thread` 对象的 `begin()` 成员函数声明为：

```
void ct_thread::begin(FunctionPtr PFN, void *X)
{
    pthread_create(&ThreadId, Attr, PFN, X);
}
```

第一个参数为函数的指针（线程的入口点）。我们传递给 `ct_thread` 对象 `begin()` 函数的另一个重要参数是 `set_server` 类的 `this` 指针。`this` 指针用于让友元函数和友元函数创建的域对象与服务器对象进行通信。将 `this` 指针用作进程间通信的机制可以不费吹灰之力！不过，因为线程为轻量级的，而且使用常规参数作为通信的一种方式，所以我们可以传递和使用 `this` 指针来进行线程间的通信。因此，`setUnion()` 成员函数创建线程，同时线程执行 `setUnion()` 友元函数。`setUnion()` 友元函数的声明如下所示：

```
void *setUnion(void *X)
{
    set_server<String> *Server;
    Server=static_cast<set_server<String> *>(X);
    mt_set<String> SafeSet(Server->Argument.A);
    Server->Argument.Result=
        SafeSet.setUnion(Server->Argument.B);
}
```

强制转换 **this** 指针

Win32、POSIX 和 OS/2 环境中的线程创建 API 都需要函数的指针。函数指针要求这个函数接受一个参数: void*。表 10-1 列出了 Win32、POSIX 和 OS/2 环境中的线程创建 API。

表 10-1 Win32、POSIX 和 OS/2 环境中的线程创建 API

环 境	函 数	参 数	描 述
POSIX	int pthread_create(pthread_t*new_thread_ID, const pthread_attr_t *attr, void* (start_funct)(void*), void *arg)	*new_thread_ID	线程的句柄
		*attr	指向属性对象的参数
		*start_funct	指定线程函数地址的参数
		*arg	线程函数的参数
		返回值	返回一个 int: 0 表示线程的成功创建, 非 0 值表示一个错误
OS/2	APIRET DosCreateThread(pptidThreadID, ppThreadAddr, ulThreadArg, ulThread Flags, ulStackSize)	pptidThreadID	被创建线程的线程 ID 的双字地址
		ppThreadAddr	指定线程函数地址的参数
		ulThreadArg	线程函数的参数
		ulThreadFlags	设置此参数来决定线程是否创建于挂起状态, 或者立即执行。如果将 0 位设置为 0, 则线程立即执行。如果将 0 位设置为 1, 则创建的线程处于挂起状态, 线程的创建者必须恢复它。如果将 1 位设置为 0, 则系统使用缺省方法来初始化线程堆栈。如果将位 1 设置为 1, 系统将预提交堆栈中的所有页
		ulStackSize	指定线程堆栈的大小
		返回值	返回一个 int: 0 表示线程的成功创建, 非 0 值表示一个错误
Win32	HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpSa, DWORD ccStack, LPTHREAD_START_ROUTINE lpStart Addr, LPVOID lpvThreadParm, DWORD fdwCreate, LPDWORD lpIDThread)	lpSa	指向定义线程安全性属性的 SECURITY_ATTRIBUTE 数据结构参数的参数
		ccStack	指定线程堆栈大小的参数
		lpStartAddr	指定函数地址的参数
		lpvThreadParm	线程函数的参数

续表

环 境	函 数	参 数	描 述
		fdwCreate	指定与线程创建有关的额外线程的参数。如果为 0，则线程立即执行。如果值为 CREATE_SUSPENDED，则线程处于挂起状态
		lpIDThread	保存线程 ID 的地址
		返回值	如果成功，返回新线程的句柄；如果不成功，返回 NULL

void* 充当线程所创建函数的一种命令行参数。程序员使用 void* 参数几乎可以给线程的主函数传递任何内容。一旦函数开始执行，它就可以将这个参数强制转换成任何一种类型。对于多线程对象，void* 将被转换成 this 指针指向的任何类型。例如，setUnion() 成员函数执行如下调用：

```
Thread.begin(::setUnion, this);
```

在这个调用中，指针指向一个 set_server<T> * 类型的对象。所以，当 setUnion() 友元函数接收到 void* 时，它将立即把这个指针转换成指向 set_server<T> 的指针。这通过 static_cast<T> 运算符来完成，如下所示：

```
set_server<String> *Server;
Server= static_cast< set_server<String> *>(X);
```

域对象与宿主对象间的连接

因为 X 是一个保存 set_server<T> 的 this 的值的 void*，我们可以转换 X，让 Server 指向正确的类型。使用 static_cast<T> 运算符，void* 可以转换成指向 T 类型对象的指针。一旦完成了转换，Server 指针现在保存宿主类 set_server 的地址。这就建立了两个线程间的通信链接。图 10-7 显示了 set_server 类与 setUnion() 友元函数间的通信关系，其中 setUnion() 友元函数在不同的线程中执行。

set_server 类是一个线程，setUnion() 成员函数位于另一个线程，而 this 指针是两个线程间的通信点。通过类型转换建立了两个线程间的通信链接后，setUnion() 友元函数创建域对象。这里的域对象是 mt_set 类。因为 setUnion() 友元函数可以访问 set_server 类，所以它可以从 set_server 类给 mt_set 类传递信息：

```
mt_set<String> SafeSet(Server->Argument.A);
```

SafeSet 对象实际上执行多线程服务器中的工作。SafeSet 对象包含对客户请求进行联合的集执行并集 (set union) 的实现。SafeSet setUnion() 成员函数的实现如下所示：

```
Template<class T> set<T,less<T>>
Mt_set<T>::setUnion(set<T,less<T>> X)
{
    less<T> Order;
    set<T,less<T>> Temp;
```

```

lock();
set_union(S.begin(), S.end(), X.begin(), X.end(),
          inserter(Temp, Temp.begin()), Order);
unlock();
return(Temp);
}

```

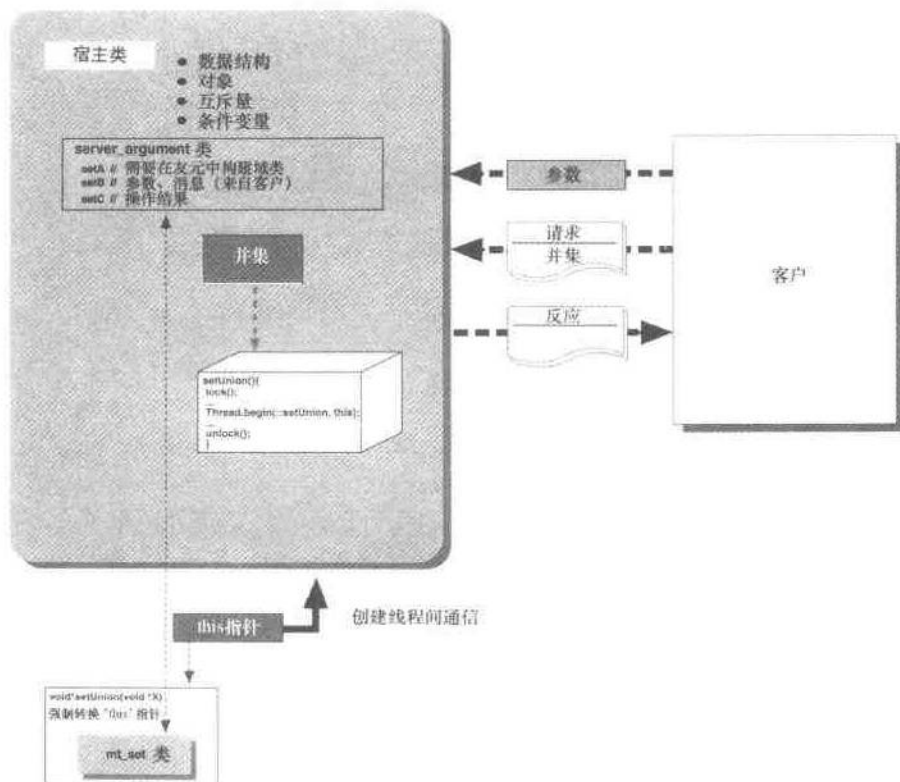


图 10-7 演示 `set_server` 类与在不同线程中执行的 `setUnion()` 友元函数之间的通信关系

SafeSet 对象保存 `set_server` 对象的 Argument 数据成员中并集的结果，如下所示：

```
Server-> Argument.Result= SafeSet.setUnion(Server-> Argument)
```

此时，`set_server` 对象可以访问这个结果，`setUnion()` 友元函数结束这个线程，`set_server` 对象可以继续它的处理。请注意，包含 `set_server` 对象的线程与 `setUnion()` 友元函数之间具有 FF (finish-to-finish) 同步依赖性。也就是说，创建 `set_server` 对象的线程不允许在执行 `setUnion()` 友元函数的线程完成之前完成。这种 FF 同步依赖性由调用 `ct_thread` 对象的 `wait()` 成员函数来强制执行。我们不希望 `set_server` 的线程在接收到并集结果前继续执行。

下面是一个简单的程序，它使用多线程集服务器来执行集 A 与集 B 之间的并集。当执行两个集之间的并集时，两个集中的成员被联合，并变成第三个集的成员：

```
#include "setserve.cpp"
```

```
#include<set.h>
#include<iostream.h>

set<String,less<String>> A;
set<String,less<String>> B;
set<String,less<String>> C;
set<String,less<String>>::iterator I;

void main(void)
{
    cout<< "start main thread "<<endl;
    A.insert("hello");
    B.insert("I'm alive");

    set_server<String> Server;
    C=Server.setUnion(A,B);
    I=C.begin();
    Cout<<"from main thread"<< endl;
    While(I !=C.end())
    {
        cout<< *I<<endl;
        I++;
    }
}
```

在这里，集 A 包含字符串“hello”。集 B 包含字符串“I'm alive”。客户请求服务器对这两个集执行并集。服务器创建一个单独的线程来执行并集，并将结果返回保存在集 C 中，集 C 现在包含两个字符串。显然，在实际中，客户可能请求更复杂的集运算，如联合到数据库，或检查一个庞大的文档查找一个单词集，或者求两个大文件的交集。

我们在这里列举一个简单的例子，是为了让读者领会多线程服务器后面的基本逻辑。

10.2.3 应用框架类

在所有能够以软件重用为目标构建的面向对象组件中，也许面向对象应用框架是最难以构建，但一旦构建后却能提供最大好处的组件。与容器类、集合类以及类型的类层次相比，应用框架一般情况下是最易于理解的面向对象类层次。应用框架属于最容易理解但最难以构建的类层次，因为它们为思想建模。也就是说，应用框架为对象交互、工作模式、动作序列、软件事务以及进程内容来建模。

其它类型的类层次通常为人物、地点或事物来建模。例如，我们谈到的大部分容器类都是为那些能够准确理解、具有离散边界的事物进行建模，如列表（list）、窗口（windows）、box、bag、表（table）、集（set）、矩阵（matrix）等等。当调用一般性类，为以上类型对象建模时，在软件中捕获它们的特征和属性相当简单。为类指定适当的抽象数据类型(ADT)是一件简单的事情。而要一般化用户与互联网交互时发生的动作序列就大相径庭了。为对象建模通常比为过程建模容易。在面向对象编程中，对象通常是人物、地点或事情。当设计面向对象应用框架时，对象是相关联

动作序列 (sequence of interrelated actions)。

所有的 C++ 程序员都熟悉过程库 (procedure library) 的概念。标准 C++ 库包括标准 C 中的过程库。程序员可以以过程库的形式使用常用多线程函数程序、数据转换以及 I/O。其中许多库具有相对应的面向对象库，它们属于所谓的类库 (class library)。

设计一个提供面向对象访问磁盘 I/O、数学函数、字符串操纵、内存管理、图像处理等的类库，与设计应用框架相比，前者更容易实现。因为一般情况下，可以基于对应的过程库进行构建。过程库对应者可以准确理解。如果正确设计了过程库，程序员就可以简单地调用过程或需要的函数，让过程完成任务，就这么容易实现。过程化库的主要问题是，当它们用于中型到大型、经常升级修改的软件结构中时，它们可能崩溃。也就是说，过程库不容易适应于改变的环境、软件的大型化。类库改进了这种状况 (Meyer, 1988)。在许多场合，通过添加面向对象特征，就可以在过程库的基础上改进得到类库。程序员构建类库时，不必从零开始，因为已经有了可以借鉴的对应过程库。对应的过程库是一个理想的起点。不过，应用框架没有对应的过程库。应用框架的概念与过程编程的思想差异太大。在过程化编程方式中，函数和过程是最高层次的一般化。使用过程途径的程序通常设计用来解决特定的问题。使用基于过程化方式设计一般性程序的思想是不切实际的。

应用框架为一般性面向对象应用。这个框架以类层次的形式捕获应用具有的动作、工作模式和处理内容的序列。应用框架用作整个应用的模式。它具体化应用具有的基础结构或骨架，而没有提供应用的细节。应用框架指定了面向对象架构中软件部分间的关系、责任和协议。应用框架以一种可用于多种不同上下文的方式来捕获软件中的这些概念。优秀应用框架的价值在于，它为不同范围的应用提供了可重用的公共特性。程序是特定问题的一般性解决方案，应用框架是某问题类别的特定解决方案。应用框架的一个典型例子是面向对象语言处理器。它所提供的特定解决方案是取得输入的语言，并将此语言翻译成输入形式。这个框架包含多个软件部分：

- 验证组件；
- 符号化组件；
- 解析器组件；
- 语法分析组件；
- 词法分析组件；
- 规则。

这些软件部分是无关的，而且可以组合形成一个非常熟悉的工作模式：

```
void main(void)
{
    language processor X;
    X.getString()
    X.validateString()
    X.parseString()
    X.lexicalGroup()
    ...
    ...
    ...
}
```

```
        X.result()  
    }
```

这个特定解决方案可以应用于不同类别的问题领域，例如：

- 编译器；
- 软件计算器；
- Web 浏览器；
- 文件传输器；
- 命令解释器；
- 自然语言处理；
- 加密/解码。

在许多不同领域重用应用框架的可能性是最强大的 C++ 类层次类型之一。事实上，应用框架的目标是成功一般化工作模式，这使得它成为最难实现的类层次之一。构建高质量的应用框架几乎需要对 C++ 中支持面向对象编程的每一种机制有一个清楚的了解。在构建应用框架前，必须准确理解封装、继承以及多态。而且，应用框架的设计需要对软件分解有一个全面的理解，因为设计者必须将框架分解成提供者实现和客户实现部分。在多线程应用框架中，框架执行的工作模式必须分解成一套线程。框架的实质是需要分解成它的一般性部分以及特定应用的部分。C++ 中通过抽象基类、虚基类、虚函数、类复合、类聚集、友元函数、模板以及新类型转换运算符来支持软件分解。

一、应用框架类层次

应用框架由相关类的集合组成。应用框架的成员函数有一个预设计的、内置的交互模式。这套相关类和预设计交互模式为程序员提供一些一般性问题的特定解决方案。正如“框架”这个词所暗示的，它只是为应用提供一个模板或一个大纲。为了让程序员最大限度地利用应用框架，程序员必须用必需的组件来填充大纲。

请想像这样的情形，假如给予用户某种类型交通工具的钢架。它有 4 个轮子、一把座椅、一台发动机以及一套转向装置。它提供了特定用户的发动机，它可以使用气体燃烧、太阳能或电能；用户可以自由选择。虽然这个交通工具有一个底盘，没有车体。它具有四轮驱动，提供了一个座椅，而且最多可支持到 8 个座椅。转向机械位于交通工具的前方。它需要一个点火系统，但没有提供特定的类型。交通工具的底盘可以根据需要伸缩。为了使用这个交通工具，它需要用户选定一种发动机类型，并提供点火系统。要让用户充分利用这个交通工具，用户必须添加足够的座椅，让底盘长度适中，而且为交通工具提供一个车体，等等。

在这个例子中，用户得到了一个交通工具框架，它可以用于解决许多不同的与交通相关的问题。如果用户的目的是从地点 A 到地点 B，那用户只需要选定一种发动机、添加适当的点火系统就可以了。交通工具框架可以完成余下的所有工作。不过，如果用户的愿望特别一些，如 6 人双层移动餐厅，则用户必须添加适当的车身和另外 5 把座椅，而且可能要适当加长底盘。如果我们的用户特别是智多谋，他可能会添加一台电力发动机以及对应的点火装置，并将这个交通工具框架进行彻底地改造，让它用作一台为工厂运输带提供动力的机器！此处重要的一点是，交通工具框架提供部分功能，而同时要求用户提供部分功能。框架提供的功能越多越好。不过，如果框架

设计得当，它将具有高度的可配置性。理想情况下，框架可以原封不动地使用，或者只需要做极少的工作就可使用，同时，还允许用户根据需要针对特定的场合修改框架。

如果我们将交通工具框架转换成一个用 C++ 组件实现的软件框架，则发动机、底盘长度以及座椅数量的选择都将由框架的构造函数来决定。点火系统将是一个纯虚函数，它必须由交通工具框架的用户在派生类中提供。交通工具车体将是继承所得后代中的子类。图 10-8 显示了交通工具框架组件与根据这个框架创建的用户交通工具对象之间的关系。

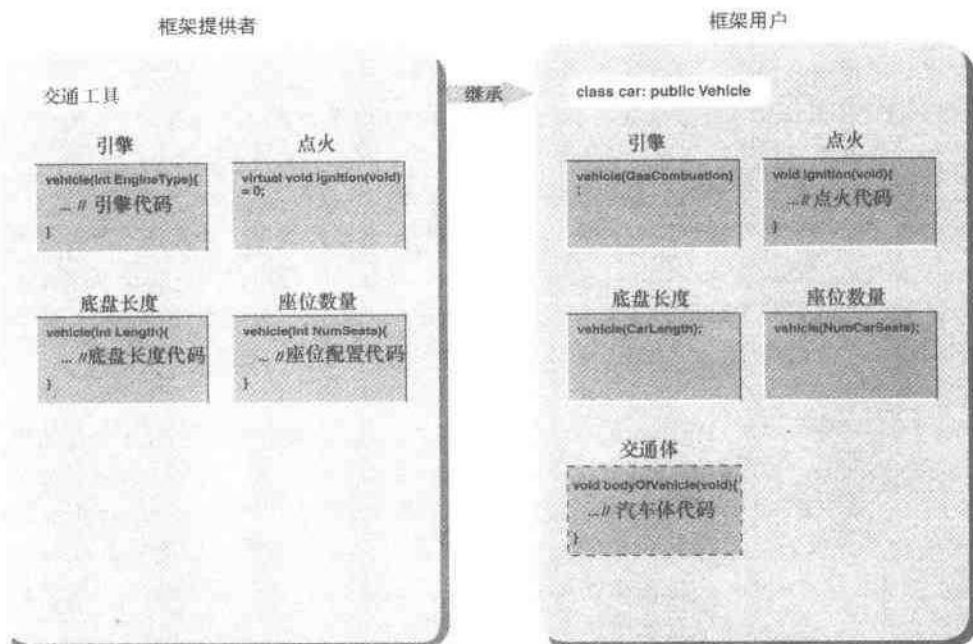


图 10-8 交通工具框架组件与从此框架创建的交通工具对象间的关系。应用框架是框架提供者与框架用户间的一种合作性契约。框架设计者提供部分框架功能（理想情况下提供大部分功能），同时框架的用户完成这个框架的功能

这个例子演示了应用框架的基本特征。应用框架是框架提供者与框架用户间的合作性软件契约。框架设计者提供框架的部分功能（理想情况下提供大部分功能），同时，该框架的用户完善框架的功能。框架设计者要指定框架中对象间的关系。框架设计者确保框架成员函数间的预设计交互模式提供一种被测试、证明、特定的一般问题类别解决方案。在交通工具框架中，排气系统、传输系统中的工作模式、如何驱动交通工具以及分解系统都由交通工具框架提供者预定义。交通工具框架只能滚动，它不会飞行或飘浮。内置动作序列是应用框架与其它类层次的区别之所在。框架提供者应该标识和提供应用必须执行的基本工作模式。框架的用户要提供所有抽象虚函数和构造函数参数的定义。

框架分解

应用框架总是至少分解为两个主要组件。第一个组件为框架设计者提供的框架部分。这个组

件包含按一般性方式捕获的特定动作序列以及对象间的关系，它表示某问题类别的一种常用解决方案。第二个组件称做 **ensemble**。**ensemble** 具体表现域知识、专家知识、规则以及特定解决方案的策略 (*The Power of Frameworks*, Taligent)。**ensemble** 由框架的用户提供。框架本身提供了应用的形式。**ensemble** 提供应用的具体内容。框架提供应用的主要控制流程。用户插入 **ensemble**，而且受框架中已存在代码的控制流程的管理。

框架中指定的动作序列由三种成员函数类型来实现。一些应用框架的成员函数为虚成员函数，应用框架成员函数的用户可以替换它们。选择权取决于用户。用户可以决定使用框架提供的功能，或者通过用户提供的成员函数覆盖这些功能。框架可以使用的第二种成员函数类型是纯虚函数，它需要框架用户提供自己的定义。纯虚函数是针对程序员必须提供的实际函数的唯一“占位符”，否则程序不能编译。纯虚函数用于完成框架内的动作序列或控制流程，它不需要指定函数实现的细节。在继承和使用框架前，所有的纯虚函数都必须在派生类中定义。

第三种函数是常规成员函数。框架使用常规成员函数来实现普通的动作序列，这些动作是派生类不需要更改或提供的。在框架中提供的常规函数越多越好。如果这些常规函数为普通工作模式的高层次一般化，而且这些工作模式在框架所应用的问题类别可以找到，在框架中提供更多这样的常规函数，尤其有利于框架的使用。应用框架在框架中使用基类的指针和引用。虚成员函数和类的引用通常是允许程序员为特定领域指定应用框架的关键。

提供 **ensemble**

应用框架的内容可以通过继承、复合或结合两者来提供。一般情况下，程序员在使用框架之前，必须继承框架类，为所有纯虚函数提供定义，覆盖必需的虚函数。当程序员通过定义纯虚函数和覆盖虚函数提供框架的内容时，程序员就是在通过继承提供应用框架的 **ensemble**。程序员还通过框架的构造函数，以及使用对象的指针或引用作为参数的所有框架成员函数为框架提供 **ensemble**。当通过构造函数、对象的指针和引用提供 **ensemble** 时，程序员就是通过复合来构建框架，这与继承相反。程序员还结合使用继承和复合来提供框架的 **ensemble**。一旦提供了 **ensemble**，框架就完成了，并表示某应用的一个具体实例。

二、一个简单应用框架

下面让我们看一个简单的应用框架。在这个例子中，我们提供一个查询处理器/解析器作为一个应用框架。解析和查询处理形成了广泛编程问题的基本解决方案。可以在查询处理器和语言解析器中发现的序列在所有类型的应用中被重复查找。如果我们可以一般化在查询处理器中发生的某些动作序列，我们就具备了应用框架的基础。对于下面要讲的简单查询处理器框架，我们集中于以下简单动作序列：

- 获取输入字符串；
- 验证输入字符串；
- 解析输入字符串；
- 判断输入字符串。

以上一般性的动作序列在每次使用框架时自动执行。应用框架的使用说明了应用框架和集合或容器类之间的一个主要区别。集合和容器类为几乎可以各种各样方式使用的一般目的的结构。

而应用框架只提供一种解决方案。虽然它可以提供广泛问题类别的解决方案，但它只表示一种解决方案。查询处理器框架只能获取字符串、验证字符串、解析字符串，然后判断字符串。它所执行的所有动作都与验证、解析和判断有关。框架不能做其它任何事。下面是简单 query_processor 类的声明：

```
typedef set<char,less<char>> VSet;

class query_processor{
protected:
    set<char,less<char>> ValidOperands;
    set<char,less<char>> ValidOperators;
    set<string,less<string>> ValidFunctions;
    set<char,less<char>> ValidCharacters;
    set<char,less<char>> ValidLetters;
    set<char,less<char>> ValidNumbers;
    string InputString;
    List<expression_component> ExpressionTokens;
    stack<vector<expression_component>> OperandStack;
    stack<vector<expression_component>> OperatorStack;
    int ValidExpression;
    char FunctionResult[10];
public:
    query_processor(string Input);
    query_processor(string Input,VSet VOperands,
                    VSet VOperators,
                    VSet VFunctions,
                    VSet VCharacters,
                    VSet VLetters,
                    VSet VNumbers);
    query_processor(void);

    int checkParenthesis(void);
    int checkOperators(void);
    int checkDecimals(void);
    int chckCharacters(void);
    virtual void evaluate(void)=0;
    int parse(void);
    string inputString(void);
    void inputString(string Inprt);
    int validExpression(void);
    double processNumber(int &Count);
    char *processFunction(int &Count);
    list<expression_component> expressionToken(void);
    void process(void);
    int validate(void);
};
```


这个 `query_processor` 框架类需要通过继承和复合来提供 `ensemble`。

通过复合的框架 (`query_processor`)

`query_processor` 框架类的主构造函数声明如下所示:

```
query_processor(string Input, VSet VOperands,
                VSet VOperators,
                VSet VFunctions,
                VSet VCharacters,
                VSet VLetters,
                VSet VNumbers);
```

它需要用户提供 6 个集:

- 有效操作数;
- 有效运算符;
- 有效函数;
- 有效字符;
- 有效数字;
- 有效字母。

这些集用于验证还要传递给构造函数的输入字符串。这个框架设计用来与这 6 个集一起工作,而且事先知道这些集将用于干什么以及它们如何关联。不过,框架不知道每个集中的内容是什么。框架只是操纵这些集,例如:

```
int query_processor::checkCharacters(void)
{
    int Count=0;
    while((Count<InputString.length()) && (ValidExpression))
    {
        if(!ValidCharacters.count(InputString[Count])){
            ValidExpression=0;
        }
        Count++;
    }
    return(ValidExpression);
}
```

`query_processor` 的 `checkCharacters()` 成员函数使用了通过框架的构造函数传递过来的 `ValidCharacters` 集,以确保输入字符串只包含有效字符。框架不知道有效字符是什么。它只知道在输入字符串中找到的每个字符也必须能在 `ValidCharacters` 集中找到,否则,输入字符串就不是一个有效表达式。框架依赖于程序员来提供表达式的合法字符。当程序员为框架提供必需的集时,程序员就使用了复合的思想来完成框架的 `ensemble`。

通过继承的框架 (`query_processor`)

`query_processor` 框架类是一个抽象基类,因为它包含一个纯虚函数:

```
virtual void evaluate(void)=0;
```

这意味着程序员不能声明 `query_processor` 类型的对象。在使用 `query_processor` 类之前,程序

员必须将 `query_processor` 当成基类派生后代类。然后，程序员必须在派生类中定义 `evaluate()` 成员函数。定义了 `evaluate()` 成员函数后，程序员就可以声明派生类类型的对象了。当程序员使用派生类并重新定义虚函数来提供框架的 `ensemble` 时，程序员正是使用继承来完成应用框架的功能。程序清单 10-1 包含 `query_processor` 框架类的定义。

程序清单 10-1 `query_processor` 应用框架的定义

```

1
2  #include "qprocess.h"
3  #include "strstrea.h"
4  #include "algo.h"
5
6
7
8  query_processor::query_processor(string Input)
9  {
10
11     char Character;
12     InputString = Input;
13     ValidExpression = 1;
14     Character = '0';
15     while(Character <= '9')
16     {
17
18         ValidNumbers.insert(Character);
19         ValidOperands.insert(Character);
20         ValidCharacters.insert(Character);
21         Character++;
22     }
23     ValidOperators.insert('*');
24     ValidOperators.insert('-');
25     ValidOperators.insert('+');
26     ValidOperators.insert('/');
27
28
29     Character = 'a';
30     while(Character <= 'z')
31     {
32         ValidLetters.insert(Character);
33         ValidCharacters.insert(Character);
34         Character++;
35     }
36     Character = 'A';
37     while(Character < 'Z')
38     {
39         ValidLetters.insert(Character);

```

```
40     ValidCharacters.insert(Character);
41     Character++;
42 }
43 ValidCharacters.insert(' ');
44 ValidCharacters.insert('.');
45 ValidCharacters.insert(',')';
46 ValidCharacters.insert('(');
47 ValidCharacters.insert('*');
48 ValidCharacters.insert('/');
49 ValidCharacters.insert('+');
50 ValidCharacters.insert('-');
51
52 ValidFunctions.insert("tan");
53 ValidFunctions.insert("sqr");
54 ValidFunctions.insert("log");
55 ValidFunctions.insert("sin");
56 ValidFunctions.insert("cos");
57 ValidFunctions.insert("acos");
58 ValidFunctions.insert("atan");
59 ValidFunctions.insert("asin");
60
61 }
62
63 query_processor::query_processor(string Input,VSet erands,
64                                 VSet VOperators,
65                                 VSet VFunctions,
66                                 VSet VCharacters,
67                                 VSet VLetters,
68                                 VSet VNumbers)
69 {
70     ValidOperators = VOperators;
71     ValidCharacters = VCharacters;
72     ValidLetters = VLetters;
73     ValidNumbers = VNumbers;
74     InputString = Input;
75     ValidExpression = 1;
76     copy(VFunctions.begin(),VFunctions.end(),
77         ValidFunctions.begin());
78 }
79
80 query_processor::query_processor(void)
81 {
82     char Character;
83     ValidExpression = 1;
84     Character = '0';
```

```
85     while(Character <= '9')
86     {
87
88         ValidNumbers.insert(Character);
89         ValidOperands.insert(Character);
90         ValidCharacters.insert(Character);
91         Character++;
92     }
93     ValidOperators.insert('*');
94     ValidOperators.insert('-');
95     ValidOperators.insert('+');
96     ValidOperators.insert('/');
97     Character = 'a';
98     while(Character <= 'z')
99     {
100         ValidCharacters.insert(Character);
101         ValidLetters.insert(Character);
102         Character++;
103     }
104     Character = 'A';
105     while(Character < 'A')
106     {
107         ValidLetters.insert(Character);
108         ValidCharacters.insert(Character);
109         Character++;
110     }
111     ValidCharacters.insert(' ');
112     ValidCharacters.insert('.');
113     ValidCharacters.insert(')');
114     ValidCharacters.insert('(');
115     ValidCharacters.insert('*');
116     ValidCharacters.insert('/');
117     ValidCharacters.insert('+');
118     ValidCharacters.insert('-');
119
120     ValidFunctions.insert("tan");
121     ValidFunctions.insert("sqr");
122     ValidFunctions.insert("log");
123     ValidFunctions.insert("sin");
124     ValidFunctions.insert("cos");
125     ValidFunctions.insert("acos");
126     ValidFunctions.insert("atan");
127     ValidFunctions.insert("asin");
128
129
130 }
```

```
131
132 int query_processor::checkParenthesis(void)
133 {
134     int LeftParen = 0;
135     int RightParen = 0;
136     int Count = 0;
137     while(({ValidExpression} && (Count < InputString.length()))
138     {
139         if(InputString[0] == ')'){
140             ValidExpression = 0;
141         }
142         if(InputString[InputString.length() -1] == '({
143             ValidExpression = 0;
144         }
145         if(InputString[Count] == ')'){
146             RightParen++;
147         }
148         if(InputString[Count] == '({
149             LeftParen++;
150         }
151         if(RightParen > LeftParen){
152             ValidExpression = 0;
153         }
154         Count++;
155     }
156     return(ValidExpression);
157
158 }
159
160 int query_processor::checkOperators(void)
161 {
162     int Count = 0;
163     while(({Count < InputString.length()}
164           && (ValidExpression))
165     {
166         // 检查开始表达式的运算符
167         if(({InputString[0] == '*' || (InputString[0] == '/')
168           || (InputString[0] == '+')){
169             ValidExpression = 0;
170         }
171         // 检查结束表达式的运算符
172         if(ValidOperators.count(InputString
173                                [InputString.length() -1])){
174             ValidExpression = 0;
175         }
176     }
```

```
175         // 检查连续运算符
176         if(ValidOperators.count(InputString[Count])){
177             if(ValidOperators.count(InputString[Count + 1])){
178                 ValidExpression = 0;
179             }
180             if(Count > 0){
181                 if(ValidOperators.count(InputString[Count - 1])){
182                     ValidExpression = 0;
183                 }
184             }
185         }
186         // 检查前面为字符的运算符
187
188         if(ValidOperators.count(InputString[Count])){
189             if(Count > 0){
190                 if(ValidLetters.count((InputString[Count - 1]))){
191                     ValidExpression = 0;
192                 }
193             }
194         }
195         Count++;
196
197     }
198     ValidOperators.insert('(');
199     ValidOperators.insert(')');
200     return(ValidExpression);
201 }
202
203 int query_processor::checkDecimals(void)
204 {
205     int Count = 0;
206
207     // 检查开始表达式的小数
208
209     if(InputString[0] == '.'){
210         ValidExpression = 0;
211     }
212     // 检查结束表达式的小数
213
214     if(InputString[InputString.length() - 1] == '.'){
215         ValidExpression = 0;
216     }
217     // 检查连续小数
218     // 以及小数前面或后续字符
219
220
```

```
221     while((ValidExpression) && (Count < InputString.length()))
222     {
223         if(InputString[Count] == '.'){
224             if((InputString[Count + 1] == '.') ||
                (InputString[Count - 1] == '.)){
225                 ValidExpression = 0;
226             }
227             if((ValidLetters.count((InputString[Count + 1])))
                || (ValidLetters.count((InputString[Count - 1])))){
228                 ValidExpression = 0;
229             }
230         }
231         Count++;
232     }
233     return(ValidExpression);
234 }
235
236 int query_processor::checkCharacters(void)
237 {
238     int Count = 0;
239     while((Count < InputString.length()) && (ValidExpression))
240     {
241         if(!ValidCharacters.count(InputString[Count])){
242             ValidExpression = 0;
243         }
244         Count++;
245     }
246     return(ValidExpression);
247 }
248
249 string query_processor::inputString(void)
250 {
251     return(InputString);
252 }
253
254 void query_processor::inputString(string Input)
255 {
256     InputString = Input;
257 }
258
259 int query_processor::validExpression(void)
260 {
261     return(ValidExpression);
262 }
263
264
```

```
265 char *query_processor::processFunction(int &Count)
266 {
267
268     strcpy(FunctionResult, "");
269     strstr FunctionName;
270
271     do{
272         if(ValidLetters.count(InputString[Count])){
273             FunctionName << InputString[Count];
274         }
275         Count++;
276     }while(ValidLetters.count(InputString[Count])
277           && (Count < InputString.length()));
278     FunctionName << ends;
279     FunctionName >> FunctionResult;
280     string F(FunctionResult);
281     if(ValidFunctions.count(F) == 0){
282         ValidExpression = 0;
283         strcpy(FunctionResult, "");
284     }
285     Count--;
286     return(FunctionResult);
287 }
288
289 int query_processor::parse(void)
290 {
291     double NumericResult = 0;
292     char FunctionName[10] = "";
293     int Count= 0;
294     while(Count < InputString.length())
295     {
296         if(InputString[Count] == ' '){
297             Count++;
298         }
299         if(ValidOperators.count(InputString[Count]) == 0){
300             if(ValidNumbers.count(InputString[Count])){
301                 expression_component TempToken;
302                 NumericResult = processNumber(Count);
303                 TempToken.isOperand(1);
304                 TempToken.value(NumericResult);
305                 ExpressionTokens.push_back(TempToken);
306             }
307         }
308         else{
309             if(ValidLetters.count(InputString[Count])){
```



```
310         expression_component TempToken;
311         strcpy(FunctionName,processFunction(Count));
312         TempToken.function(FunctionName);
313         TempToken.isOperator(1);
314         TempToken.isFunction(1);
315         TempToken.op('@');
316         ExpressionTokens.push_back(TempToken);
317     }
318 }
319 }
320 else{
321     expression_component TempToken;
322     TempToken.op(InputString[Count]);
323     TempToken.isOperator(1);
324     ExpressionTokens.push_back(TempToken);
325 }
326 }
327 Count++;
328 }
329 }
330 return(ValidExpression);
331 }
332 }
333 }
334 }
335 }
336 double query_processor::processNumber(int &Count)
337 {
338     double TempNumber;
339     int DecimalCount = 0;
340     stringstream Number;
341     int NotFinish = 1;
342 }
343 do{
344     if(InputString[Count] == '.'){
345         DecimalCount++;
346     }
347     if((ValidNumbers.count(InputString[Count]))
348         || (InputString[Count] == '.')){
349         Number << InputString[Count];
350         Count++;
351     }
352     else{
353         NotFinish = 0;
354     }
```

```
355         )while((Count < InputString.length())
           && (DecimalCount <=1) && (NotFinish));
356         if(DecimalCount <= 1){
357             Number << ends;
358             Number >> TempNumber;
359         }
360         else{
361             TempNumber = 0;
362             ValidExpression = 0;
363         }
364         Count--;
365         return(TempNumber);
366     }
367
368     list<expression_component>
           query_processor::expressionToken(void)
369     {
370
371         return(ExpressionTokens);
372     }
373
374     int query_processor::validate(void)
375     {
376
377         if(checkParenthesis() == 0){
378             return(0);
379         }
380         if(checkDecimals() == 0){
381             return(0);
382         }
383         if(checkOperators() == 0){
384             return(0);
385         }
386         if(checkCharacters() == 0){
387             return(0);
388         }
389         return(1);
390     }
391
392     void query_processor::process(void)
393     {
394         if(validate()){
395             parse();
396             evaluate();
397         }
398     }
```

query_processor 框架提供的基本普通动作序列由 process() 成员函数来捕获, 它们有: validate()、parse() 以及 evaluate()。

为了让程序员使用 query_processor 类, 程序员必须提供定义纯虚函数的后代类。前面讲过应用框架的一个强大特征是它对广泛问题类别提供一般性解决方案。在这里, 我们使用 query_processor 框架类来帮助我们实现一个软件计算器。

完善 query_processor 框架

首先, 我们使用继承来为框架类添加 calculator 类的功能。通过常规类继承, 这些功能也相应添加到了后代类中。通过应用框架, 后代类的功能被添加到框架类:

```
class calculator:public query_processor{
protected:
    double Result;
    double processOperator(expression_component Operation);
public:
    calculator(string Input);
    calculator(string Input,VSet VOperands,
                VSet VOperators,
                VSet VFunctions,
                VSet VCharacters,
                VSet VLetters,
                VSet VNumbers),

    calculator (void);
    double result(void);
    void processCloseParenthesis(void);
    void evaluate(void);
};
```

calculator 类按两种方式为 query_processor 框架提供 ensemble。第一种方式, 利用 calculator 构造函数提供 query_processor 使用的 6 个集。提供这些集完成了计算器的复合责任。为了完善 ensemble, calculator 定义了 evaluate() 成员函数。

```
void calculator:: evaluate(void)
{
    list<expression_component>::iterator X;
    X=ExpressionTokens.begin();
    expression_component Temp;
    expression_component Stacktop;
    expression_component Answer;
    double Value;
    while(X !=ExpressionTokens.end())
    {
        Temp=*X;
        if(Temp.isOperand()){
            OperandStack.push(Temp);
        }
    }
}
```

```

else{
    if(Temp.op()=='('){
        OperatorStack.push(Temp);
    }
    else{
        if(Temp.op()==' '){
            processCloseparenthesis();
        }
        else{
            if(OperatorStack.empty()){
                OperatorStack.push(Temp);
            }
            else{
                Stacktop=OperatorStack.top();
                If(Temp.opType()>Stacktop.opType()){
                    OperatorStack.push(Temp);
                }
                else{
                    OperatorStack.pop();
                    Value=processOperator(Stacktop);
                    Answer.isOperamd(1);
                    Answer.value(Value);
                    OperandStack.push(Answer);
                    OperatorStack.push(Temp);
                }
            }
        }
    }
}
x++;
}
while(!OperatorStack.empty())
{
    Temp=OperatorStack.top();
    OperatorStack.pop();
    Value=processOperator(Temp);
    Temp.isOperand(1);
    Temp.value(Value);
    OperandStack.push(Temp);
}
Answer=OperandStack.top();
OperandStack.pop();
Result=Answer.value();
}

```

calculator 类的 evaluate()成员函数，与提供 6 个集的构造函数一起完成了 query_processor 框

架的 ensemble。一旦完成了框架，我们就可以声明 calculator 类型的对象：

```
void main(void)
{
    String Input("(8-5)*2 + sin(4-2)");
    calculator Calculator(Input);
    Calculator.process()
}
```

然后，我们发送计算器一条 process()消息并继续。为计算器所做的大部分工作都在 query_processor 应用框架中完成。query_processor 框架包含即将执行的主要动作序列。将框架分解成主要组件后，下一项工作就是准备让它适应于多线程环境。我们将在第 11 章探讨 query_processor 框架和多线程处理。

类行为和线程处理

隐含假设是，系统将来的行为不会受到任意小变化的重大影响。

Quantum Physics: Illusion or Reality

——Alastair Rae

在使用面向对象组件构建多线程应用程序时，程序员必须考虑的重要两点是：

- 构建应用程序需要的组件；
- 这些组件在多线程环境中的行为。

我们已经讨论了互斥量对象、同步对象以及线程对象。这些对象组成了多线程应用程序的主要构建块。不过，一个完整的面向对象应用程序需要域对象、接口对象、应用对象（utility object）以及其它支持对象。我们必须确定这些对象在多线程环境中的行为方式。让我们首先来看看对象是如何生存于两个线程之中的。

11.1 线程、对象和作用域

C++语言中的对象可以有 4 种类型的作用域（scope）。作用域是程序文本中的一个区域，只有在此区域才能合法访问对象。对象具有的 4 种作用域类型为：

- 局部作用域（local scope）；
- 函数作用域（function scope）；
- 文件作用域（file scope）；
- 类作用域（class scope）。

表 11-1 描述了 C++语言中对象可能具有的每种类型作用域

理解这些不同类型作用域之间的区别以及包含多个线程的程序使用它们的方式，这一点对于程序员很重要。当把程序分成多个进程时，每个进程有自己的文本（text）、数据（data）和堆栈

片断 (stack segment)。每个程序都有自己的堆 (heap)。为了让某个进程访问属于另一个进程的数据值或命令, 这个进程必须进行进程间通信。典型情况下, 进程间通信机制位于每个进程的外部, 并且充当进程间数据流的通道。除了由传递命令行参数、子进程从父进程继承文件描述符产生的单向通信外, 没有切实的办法让一个进程访问另一个进程的数据片断、文本或堆栈片断。事实上, 操作系统的一个主要功能是防止一个进程访问另一个进程的地址空间。

表 11-1 C++语言中对象可能具有的每种类型作用域

作用域类型	描 述
局部	只能在声明它所在的块内部使用的名字。它也可以在声明点后它所包含的块内使用。函数形式参数的名字具有此函数最外部块的作用域
函数	只具有函数作用域的标签。标签可以在声明它所在函数的任何地方使用
文件	如果它在块和类外部声明, 则为具有文件作用域的名字。它可以用于声明点后声明它所在的事件单元内。具有文件作用域的名字为全局性的
类	<p>类成员的名字对于它的类是局部的。它只能用于:</p> <ul style="list-style-type: none"> • 在此类的成员函数中 • 当应用于它的类的某对象时, 能用于(.)运算符之后 • 当用于指向它的类或派生类的某对象的指针时, 能用于它的类的派生类中(>)运算符之后, 或者应用于它的类或派生类时, 能用于作用域分辨率运算符(::)之后 <p>在友元函数中声明的名字首先属于全局作用域。对于在返回值或参数类型中声明的名字类也是如此。</p>

11.1.1 连接与作用域

第3章已经讨论过, 线程共享数据片断。一个线程可以访问另一个线程的堆栈片断。当一个程序创建多个线程时, 每个线程都可以访问创建它们的进程的数据片断、堆栈片断、代码片断以及堆。当多个线程访问一个对象时, 作用域、对象以及对象的连接 (linkage) 决定了线程是否可以直接访问这个对象; 或者这个对象是否必须通过参数从一个线程传递到另一个线程。对象的名字有两种类型的连接。因为 C++支持单个程序包含多个源文件的概念, 所以产生了连接问题。程序的每个源文件称做一个翻译单元 (translation unit)。在一个翻译单元中具有文件作用域的对象可以具有内部连接 (internal linkage) 或外部连接 (external linkage)。如果文件作用域对象具有内部连接, 只能在声明它的翻译单元中访问它。如果文件作用域对象具有外部连接, 它可以被其它翻译单元中的函数和对象访问。

当文件作用域对象具有外部连接时, 它可以被创建该对象的进程中的任何线程自由访问, 不需要传递参数。文件作用域对象和外部连接对于整个进程及其所有线程是全局性的。另一方面, 文件作用域对象和内部连接可以被同一翻译单元的函数部分自由访问, 不必进行参数传递。不过, 为了被其它翻译单元中的函数访问, 该对象必须作为一个参数进行传递。就像程序可以执行在多

一个翻译单元中定义的函数和过程一样，线程也可以执行在多个翻译单元中定义的函数和过程。对象作用域、可见性以及连接规则应用于线程的方式与应用于主程序的方式一样。如果线程在执行一个函数，这个函数就容易访问位于函数局部翻译单元内的对象以及具有文件作用域和外部连接的对象。为了让线程的函数访问具有文件作用域和内部连接的对象，必须通过参数将对象传递给线程的函数。如果对象不具有文件作用域，而且不在线程执行的函数的内部，则必须通过参数将对象传递给线程的函数。表 11-2 列出了线程需要参数来访问对象的条件，以及线程可以通过参数自由访问对象的条件。

表 11-2 线程需要参数来访问对象的条件，以及线程可以通过参数自由访问对象的条件

作用域	外部	内部
文件	无必需参数	作为参数传递
局部	无	作为参数传递
类	无	作为参数传递
函数	无	作为参数传递

从表 11-2 可见，至少有一种方法可以从进程内的任一个线程访问任一个对象。这种访问可以是单向的，即表示接收线程只能读对象，而不能更改它。这就是通过值传递对象的情况，这与通过引用传递相反。同样，如果将对象声明为 `const`，情况也是这样。如果对象通过引用或指针来传递，则对象充当线程间的通信链接，而且线程间的通信是双向通信。这意味着线程 A 对象所作的任何修改都立即影响线程 B，同时，线程 B 所做的修改也立即影响到线程 A。

11.1.2 线程和类作用域

一般而言，如果对象具有外部文件作用域，它们可以被任何线程访问，而不必传递参数。对于类作用域也是这样吗？数据成员对于它的类具有局部性，只能被类的成员访问。当对象的成员函数创建了一个线程时，情况又如何呢？既然线程是由对象创建的，这个线程能自动拥有访问对象的数据成员的权限吗？线程访问仍然受类作用域与连接的限制。如果声明对象具有文件作用域和外部连接，则线程可以访问对象的所有公有成员函数，不必借助其它特殊手段。如果声明对象具有文件作用域和内部连接，而且线程执行的函数与对象位于相同的翻译单元，则线程可以访问对象，也不必借助其它特殊手段。在其它任何场合，必须在包含原始对象的线程与对象成员函数创建的线程间进行参数传递。记住，尽管成员函数创建了线程，但成员函数和线程并没有父-子关系。由成员函数创建的线程在访问对象的内部数据成员上没有任何特殊优先权。一旦创建了线程，对象访问就由对象作用域和连接所控制。

11.2 同步关系和对象成员函数

请回忆单个进程中任何两个线程间的 4 种基本同步关系。这些关系适用于从多个线程执行函

数的类的成员函数间的关系，也适用于在一个进程中执行多个线程的多个类成员函数间的关系。表 11-3 列出了 4 种基本同步关系及其描述。

表 11-3 4 种基本同步关系及其描述

基本同步关系	描 述
SS (start-start)	线程 B 启动后线程 A 才能启动。线程 B 可以在线程 A 启动的同时或之后启动，但决不能在其之前启动
FS (finish-to-start)	线程 A 完成一定的任务后，线程 B 才能启动
SF (start-to-finish)	线程 B 完成一定的任务后，线程 A 才能启动
FF (finish-to-finish)	线程 B 完成后，线程 A 才能完成。线程 A 可以在线程 B 完成后完成，但不允许线程 A 在其之前完成

在一个进程内，对象与多个线程可能具有 4 种基本同步关系，如下所示：

- 对方被需要同步的多个线程访问。
- 对象被分解成需要同步的多个线程。
- 对象被分解成需要同步的多个线程，同时，对象也能被需要同步的多个线程访问。
- 对象只能被单个线程访问，而且不能分解成多个线程。

当把对象的功能分解成多个线程时，这些线程一般情况下需要同步。当多个线程修改它们的宿主对象时，这些线程需要同步以防止对象成员函数间的竞争条件。当多个线程一起工作完成一个共同的目标时，这些线程必须同步，让这些线程按适当的顺序完成各个步骤。这意味着在对象的线程间至少有两种依赖性关系：通信依赖性（communication dependency）与同步依赖性（synchronization dependency）。当对象创建的线程试图并发修改对象的临界区时发生通信依赖性关系。为了防止线程破坏对象的临界区，这些线程将通过互斥量来通信。当多个线程为共同的目标而工作，每个线程解决问题的一部分，此时就发生同步依赖性关系。每个部分的执行必须按正确的序列进行，以便对象从整体上协调完成它的工作。同步化对象的线程所执行的任务，线程将使用条件变量、等待函数或事件互斥量。

表 11-4 显示了满足对象所创建线程间通信依赖性关系的机制，以及用于满足可以在这些线程间发生的同步依赖性关系的机制。

表 11-4 依赖性类型与线程间机制

依赖性类型	线程间机制
通信	互斥量
同步	事件互斥量 条件变量 等待函数

这些机制应当是对象的一部分。它们可以通过继承或复合包含在对象中。在所有可能的地方，

每个对象都必须包含它的同步机制。

一个多线程化的线程框架实例

上一章介绍了应用框架 (application framework) 的概念。我们使用 `query_processor` 框架来实现一个简单的计算器。这个计算器可以接受一个包含算术运算和函数的字符串表达式, 例如:

```
(3+sin(1.53)+tan(1.01)/4)
```

并且返回一个数字结果, 它表示字符串所包含表达式的值。计算器利用 `query_processor` 框架的功能将字符串解析成适当的符号。计算器 `query_processor` 具有的流程控制来验证和解析字符串。处理化计算器的主要成员函数, 可以将计算器转换成一个简单的计算器服务器。我们使用第 10 章用于生成多线程集服务器的相同技术来线程处理本计算器。计算器服务器至少需要 6 个组件:

- 宿主类;
- 线程类;
- 互斥量和事件类;
- 友元成员函数
- 域类;
- 强制转换基本元素。

服务器需要一个宿主类, 我们称做 `threaded_calculator`。服务器创建的线程由那些从接口类创建的线程对象来实现, 这些接口类封装了相应操作系统的线程处理 API 函数。为了防止竞争条件并允许同步, 计算器服务器需要互斥量和事件类。服务器还需要域类, 我们为它起名为 `mt_calculator`。`mt_calculator` 类是针对 `calculator` 类的一个接口类。`mt_calculator` 提供了由 `calculator` 类实现的高层次事务。`mt_calculator` 类为 `calculator` 类提供 `locking()` 和 `unlocking()` 服务。`calculator` 类不是线程安全的, 为了让它成为线程安全, 我们在 `mt_calculator` 类中封装它。图 11-1 显示了多线程计算器服务器中组合在一起的各个组件。

虽然 `mt_calculator` 类是线程安全的, 但它不是多线程的; 我们使用 `threaded_calculator` 类来提供多线程计算器服务。由 `threaded_calculator` 对象创建的线程用于执行 `threaded_calculator` 类的友元函数。`threaded_calculator` 类将 `this` 指针传递给该友元函数。`threaded_calculator` 类与 `mt_calculator` 类的类关系如图 11-2 所示。

一旦友元函数拥有了 `this` 指针, 它们将使用 `static_cast<T>` 运算符将 `this` 指针转换成指向 `threaded_calculator` 对象的指针。这允许友元函数访问 `threaded_calculator` 的数据组件。这种强制转换也在创建 `threaded_calculator` 对象的线程与 `threaded_calculator` 对象创建的线程间建立了通信。我们使用这 6 个基本组件来构建 `threaded_calculator` 类:

```
class threaded_calculator:private mutex{
private:
    String InputString;
    double Result;
    list<expression_component> Tokens;
public:
    threaded_calculator(void);
    friend void evaluate(void *X);
    friend void parse(void *X);
```

```
double evaluate(String Input);
list<expression_component> parse(String Input);
};
```

我们利用 C++ 的函数重载，根据 `threaded_calculator` 类的对应名字给友元函数命名。这样使得友元函数更容易与它的对应成员函数相关联。`threaded_calculator` 类中的每个友元函数代表一个单独的线程。所以，`threaded_calculator` 的功能被分解成 3 个线程。`threaded_calculator` 将存在于创建它的线程中。`threaded_calculator` 类的 `evaluate()` 和 `parse()` 成员函数在执行时各自创建一个单独的线程。让我们检查 `evaluate()` 成员函数来看看是如何实现的：

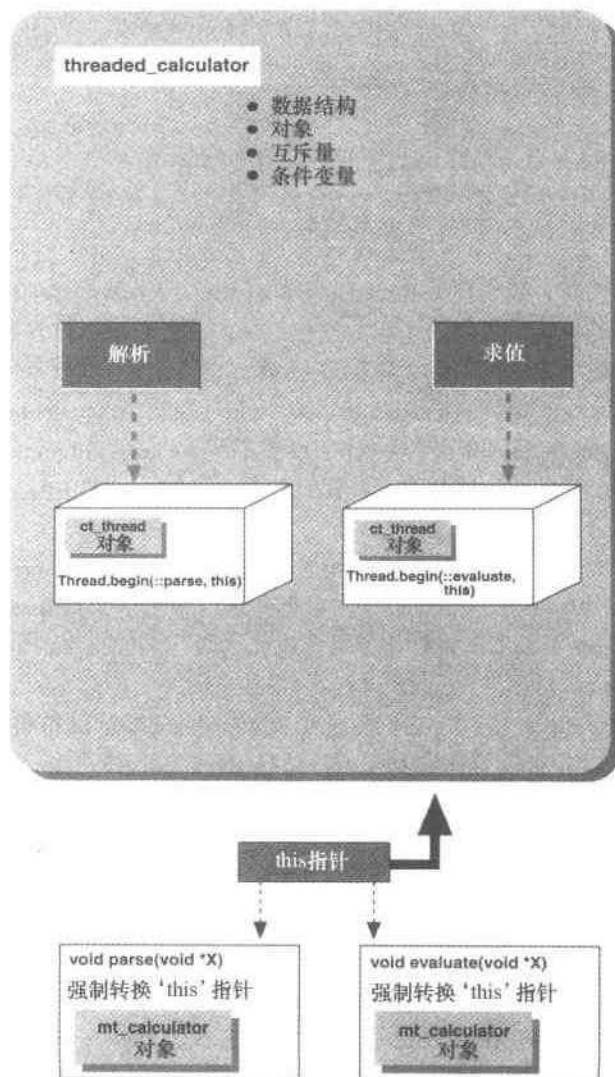
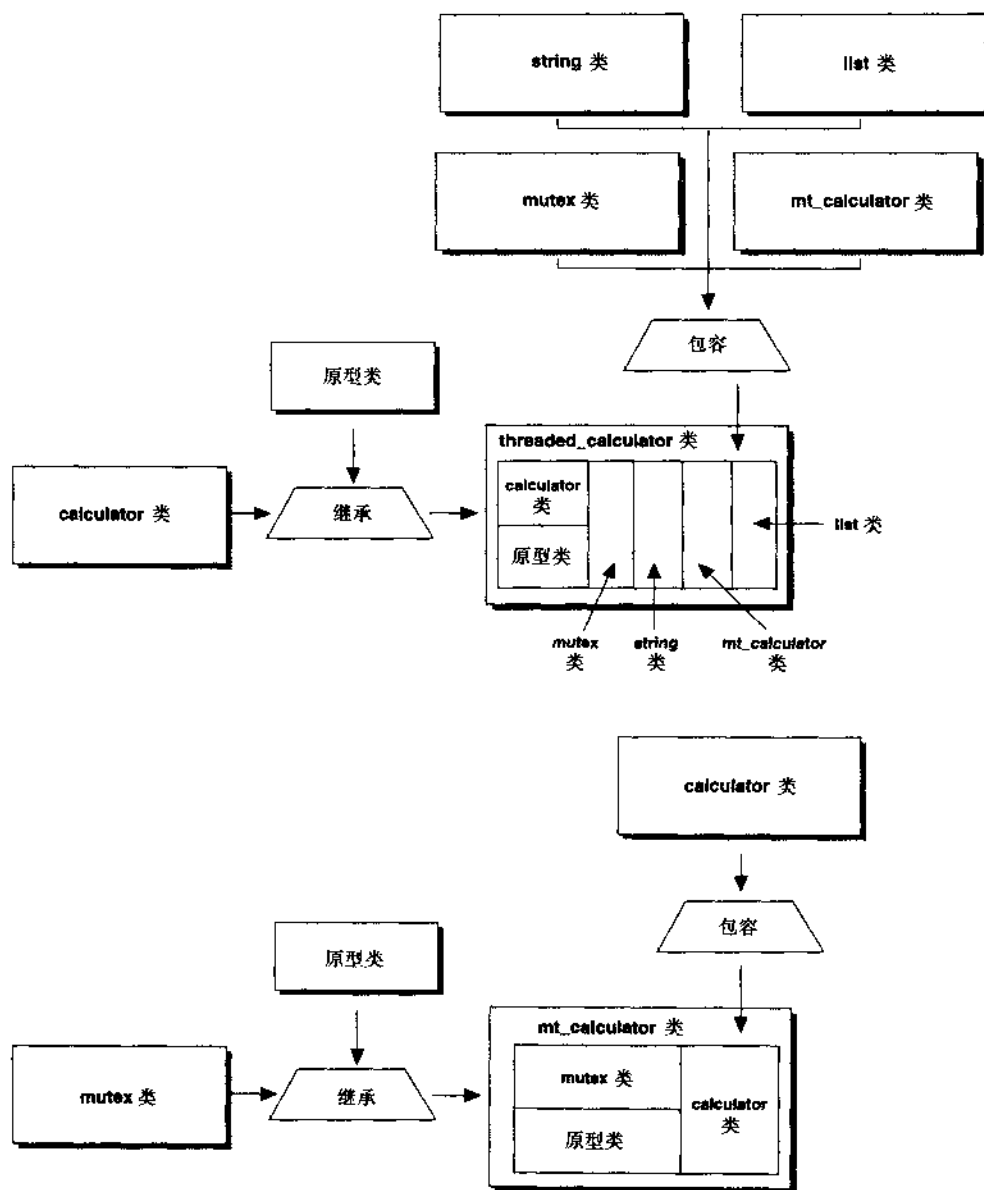


图 11-1 多线程计算器服务器的组件

图 11-2 `threaded_calculator` 与 `mt_calculator` 类之间的类关系

```
double threaded_calculator::evaluate(string Input)
{
    ct_thread Thread;
    double Temp;
    lock();
```

```

Thread.begin(::evaluate,this);
Thread.wait();
Temp=Result;
unlock();
return(Temp);
}

```

`evaluate()`函数创建一个 `ct_thread` 类型的对象, 名叫 `Thread`。这个函数然后调用 `lock()`函数获取它的互斥信号量。如果信号量已经被另一个线程占有, 则 `evaluate()`阻塞, 直到释放了锁定为止。一旦 `evaluate()`拥有了信号量的占有权, 它就启动一个线程向线程传递它的 `this` 指针。这个线程从 `evaluate()`友元函数开始执行。线程启动后, `threaded_calculator` 的 `evaluate()`成员函数停止, 必须等待 `evaluate()`友元函数的完成。创建 `threaded_calculator` 对象的线程以及 `threaded_calculator` 对象创建的线程具有 FF (finish-to-finish) 关系。事实上, `evaluate()`和 `parse()`成员函数与它们创建的线程都具有 FF 关系。这意味着, 不允许成员函数在友元函数完成前完成。`evaluate()`和 `parse()`成员函数相互间存在通信依赖性, 与它们创建的线程间存在同步依赖性。图 11-3 显示了 `evaluate()`和 `parse()`成员函数与 `evaluate()`和 `parse()`友元函数间的线程依赖性。

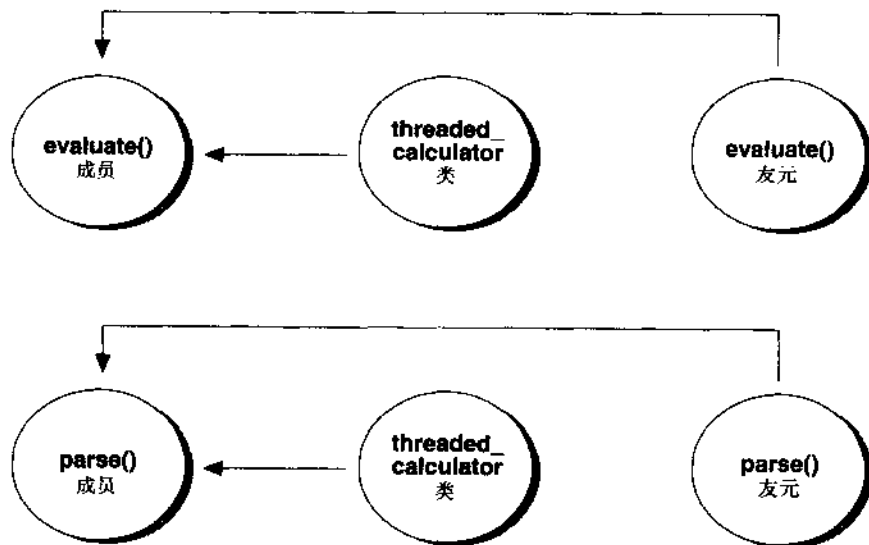


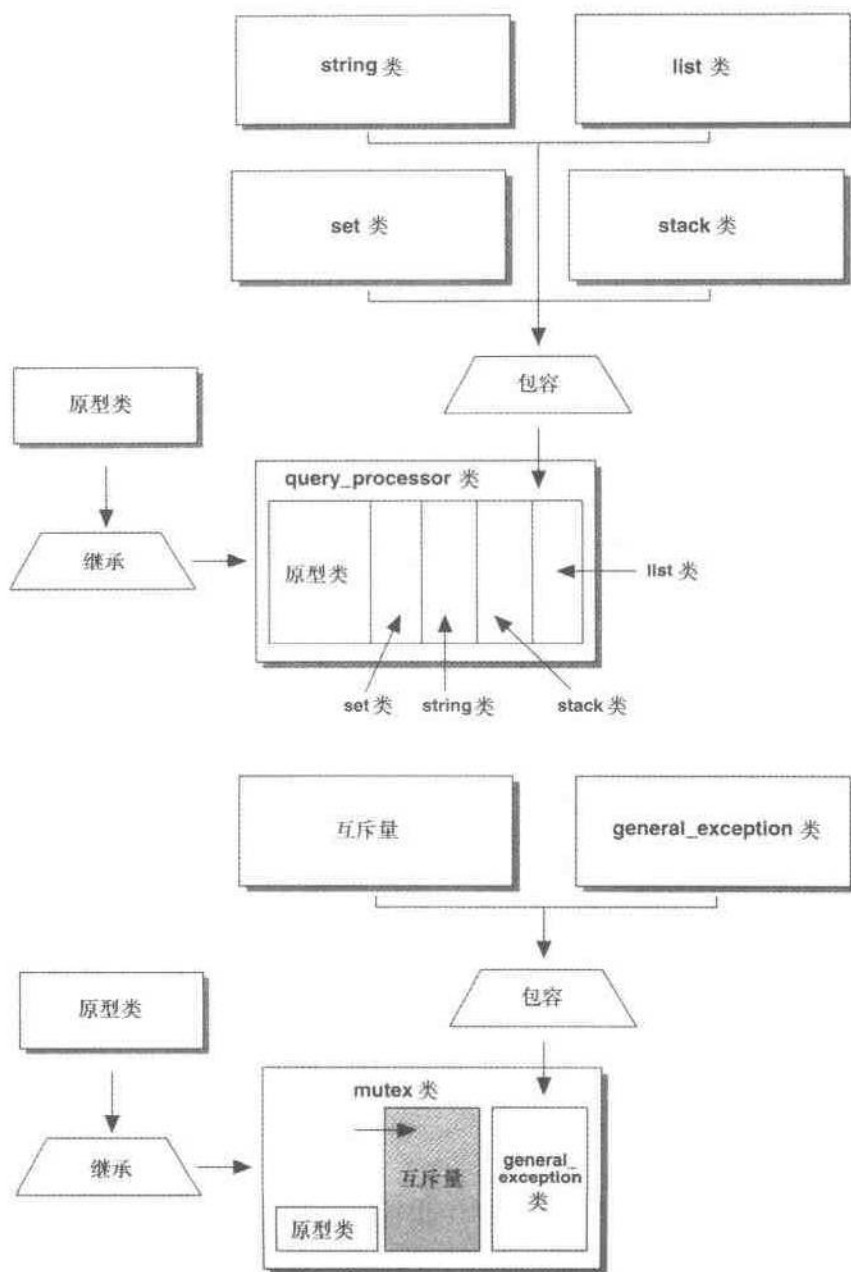
图 11-3 `evaluate()`成员函数与 `evaluate()`友元函数间、`parse()`成员函数与 `parse()`友元函数间的线程依赖性图

`evaluate()`和 `parse()`成员函数各自依赖于对 `threaded_calculator` 对象的临界区的访问。所以, 它们将互斥量用作同步机制。`evaluate()`成员函数依赖于 `evaluate()`友元函数得到计算的结果。在 `evaluate()`成员函数给计算器的客户返回结果前, `evaluate()`友元函数必须首先计算出结果并完成它的处理部分。这体现了 `evaluate()`成员函数与 `evaluate()`友元函数间的 FF 关系。

请注意, 通过提供接口类层, 我们给 `calculator` 类提供了线程处理能力。通过给 `calculator` 类提供线程化服务, 我们也给相应的 `query_processor` 框架类提供了线程化服务。我们之所以采取这种简单的方式来多线程处理, 是因为许多类并不是最先设计用于多线程环境。因此, 我们需要一

种技术让我们在多线程环境中安全使用这些类。使用接口类帮助我们进行增量多线程处理是达到目的的方式之一。图 11-4 显示了多线程计算器服务器所涉及的所有类的类关系。

线程接口类、互斥接口类、域类以及应用框架的类层次说明了如何将 C++ 组件用作多线程架构基石的。程序清单 11-1 包含 `threaded_calculator` 类的定义。



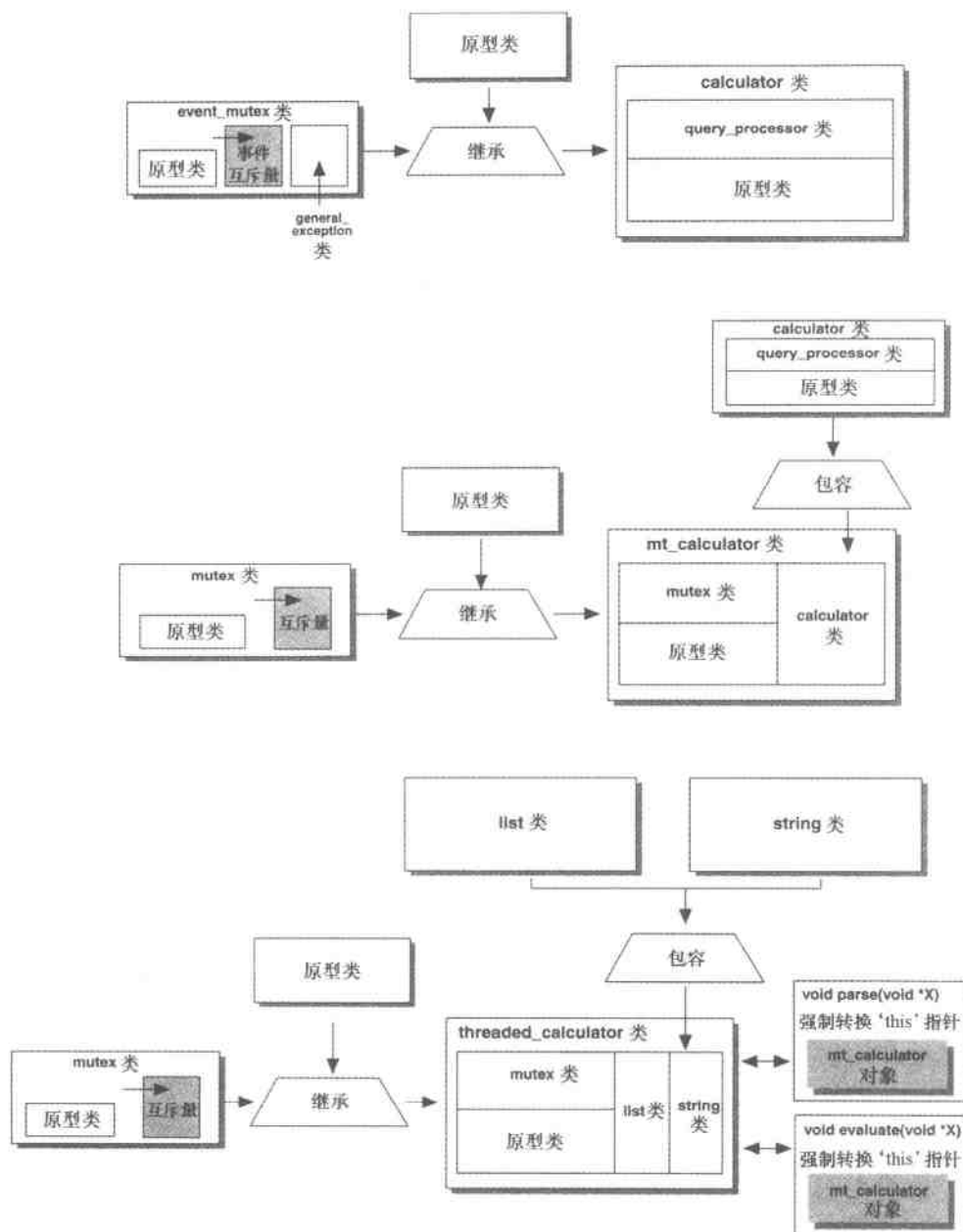


图 11-4 多线程计算器服务器所涉及的所有类的类关系图

程序清单 11-1 threaded_calculator 的定义

1
2

```
3
4  #include <calculat.h>
5  #include <ctthread.h>
6
7
8  threaded_calculator::threaded_calculator(void)
9  {
10     Result = 0;
11 }
12
13 double threaded_calculator::evaluate(string Input)
14 {
15     ct_thread Thread;
16     double Temp;
17     lock();
18     InputString = Input;
19     Thread.begin(::evaluate,this);
20     Thread.wait();
21     Temp = Result;
22     unlock();
23     return(Temp);
24 }
25
26 list<expression_component> threaded_calculator::parse
    (string Input)
27 {
28     ct_thread Thread;
29     list<expression_component> Temp;
30     lock();
31     InputString = Input;
32     Thread.begin(::parse,this);
33     Thread.wait();
34     Temp = Tokens;
35     unlock();
36     return(Temp);
37 }
38
39 void evaluate(void *X)
40 {
41     threaded_calculator *Calc;
42     Calc = static_cast<threaded_calculator *> (X);
43     mt_calculator Calculator;
44     Calc->Result = Calculator.evaluate(Calc->InputString);
45 }
46
47 void parse(void *X)
```



```

48  {
49      threaded_calculator *Calc;
50      Calc = static_cast<threaded_calculator *> (X);
51      mt_calculator Calculator;
52      Calc->Tokens = Calculator.parse(Calc->InputString);
53  }

```

`thread_calculator` 服务器对象的客户不必担心在多线程应用中编写保护 `calculator` 的代码。所有的线程处理、锁定处理以及取消锁定处理都由对象的提供者完成。用户并不知道 `threaded_calculator` 类使用的互斥量和线程 API 实现细节。`threaded_calculator` 对象封装了特定线程功能。为了使用 `threaded_calculator` 对象，客户可以编写如下代码：

```

#include "calculate.h"
#include <iostream.h>
#include <String.h>

void main(void)
{
    threaded_calculator X;
    String Input("(3+6) * sin(1.1) ");
    cout<<X.evaluate(Input);
    cout<<endl;
}

```

以上代码创建一个计算器服务器，它使用多个线程来满足使用这个线程化计算器功能的客户请求。

11.3 在多线程环境中构建和析构对象

我们在本书中所看到的大部分类通过成员函数显式调用 `lock()` 和 `unlock()`，而不是使用类的构造函数和析构函数。在构造函数中放置锁定机制、在析构函数中放置取消锁定常常是有用的。这确保了创建对象之时，它的临界区自动受到锁定。同时，当销毁对象时，被对象占有的互斥量也自动释放。互斥量自动释放是一种防止死锁的重要技术。通过在析构函数中放置释放或取消锁定功能，避免了许多可能导致死锁的情况出现。不过，仍然有一些情况可能导致绕过对象的析构函数。在这种情况下，锁定保持有效，等待锁定的其它所有对象都将无限等待。这种情形创建了一种称做无限延迟（indefinite postponement）的条件。

11.3.1 `exit()` 和 `abort()`

使用线程时请记住 `exit()` 和 `abort()` 这样函数的行为。当应用程序中发生错误时，程序员通常求助于这些函数之一。如果应用程序拥有一个共享的成员、命名互斥量、命名信号量或条件变量，程序员必须小心选择使用哪一个调用。如果不存在结构化的异常处理，则推荐使用 `exit()`。调用这个函数时，系统尝试清空缓冲器、释放资源，并在可能的地方调用析构函数。如果析构函数中存在锁定或释放机制，则调用 `exit()` 通常会让挂起的析构函数执行。另一方面，通过 `abort()` 调用，所

以执行都被取消。如果存在还没有调用的析构函数, 执行 `abort()` 请求后它们不被调用。如果这些析构函数包含对一些当前占有互斥量的取消锁定或释放机制, 则应用程序将退出并保持对互斥量的占有权, 其它所有等待该互斥量的进程或线程可能被无限延迟。在多线程环境中强烈推荐使用 C++ 异常处理机制。在程序失败或抛出异常时, 结构化异常处理让程序员有机会决定通过同步机制如何应付。

11.3.2 构造函数和 SS 关系

当在同一个进程中创建多个线程时, 它们可能并发执行、同时执行, 也可能异步执行。虽然这些术语在不严格的情况下可以交替使用, 但实际上它们存在区别。当两个线程并发执行时, 它们在同一时间段内执行, 不一定在同一时刻执行。例如, 假如有两个线程, 一个线程为一个生产者线程, 另一个线程为消费者线程。它们可能共享一个队列。生产者在队列中放入信息, 消费者从队列中删除信息。有一个需要 2 分钟完成的事务; 在这 2 分钟之内, 两个线程都在执行, 轮流访问队列。对于生产者放弃队列控制并交给消费者之前对队列的访问频率没有作出要求。同样, 在消费者放弃队列控制并交给生产者之前, 也允许消费者对队列进行不同数量的访问。关键点在于, 在这 2 分钟之内, 两个线程都是激活的, 一般要轮流访问某些共享资源。这样的线程就认为是并发执行的。在单个处理器系统或只有少量处理器的系统中进行多线程处理通常出现这种情形。

当多个线程同时执行时, 则是指它们在同一时刻执行。显然, 在单个处理器系统中, 这是不可能的。单个处理器系统中只能模拟同时执行。不过, 多处理器系统可以让某进程中的多个线程在同一时刻执行。当设计多线程应用程序时, 最好假定线程允许同时执行, 与简单的并发执行不一样。如果应用程序设计操作于线程可以执行于多处理器的环境中, 则应用程序将运行在一种线程只并发执行的环境中。不过, 反之则不一定成立。

如果多个线程异步执行, 它们可能并发执行、同时执行, 也可能会依次执行。对于异常执行, 不能保证多个线程的执行方式。在异步执行的进程中, 线程执行的顺序得不到保证, 第一个线程可能完整执行, 然后再执行第二个线程。在同一进程的下次运行中, 第二个线程可能完整执行, 然后再执行第一个线程。另一方面, 两个线程可能并发执行, 轮流访问共享资源。当使用异步执行时, 并不能确保单个进程中的多个线程将采取并发或同时执行。

SS (start-to-start) 关系

计划为多线程应用程序设计交互对象的程序员应当留心并发执行线程、同时执行线程以及异步执行线程间的区别。在多线程应用程序的对象间通常存在 SS (start-to-start) 关系。如果我们按这种方式在构造函数中锁定互斥量, 我们必须留意具有 SS 关系的对象。这意味着直到构建另一个对象后, 这个对象才能被构建。如果这些对象处于分离的线程中, 那么程序员可以使用条件变量或事件互斥量来管理对象间的 SS 关系。

再以生产者-消费者为例子, 将生产者对象分配给线程 A, 将多个消费者对象分配给其它多个线程, 名叫线程 B、线程 C 和线程 D。如果主线程创建了 4 个线程 A、B、C 和 D, 则哪一个线程将首先执行? 如果线程 B、C 或 D 在线程 A 启动前启动, 则首先构建消费者对象, 并试图访问一个空队列, 因为线程 A 中的生产者对象还没有被构建。我们说线程 B、C 和 D 与线程 A 具有 SS 关系。也就是说, 线程 A 中的生产者对象没有构建和启动之前, 线程 B、C 和 D 的对象不能

启动它们的处理。所以，在生产者或服务器对象的构造函数中放置一个条件广播或事件互斥量常常有用。同样，消费者对对象的构造函数也包含事件等待，在它们接收到来自生产者对象的广播之前，它们将一直阻塞。一旦构建同步后，生产者对象立即开始“生产”，消费者对象立即开始“消费”，在这种场合，这种同步类型特别有用。这也适用于与其它线程中对象具有 SF (start-to-finish) 和 FS (finish-to-start) 关系的对象。通过确保每个对象包含自己的事件互斥量可以管理这些关系。请回忆前面讲过的 event_mutex 接口类：

```
class event_mutex{
protected:
    HEV EventMutex;
    char MuxName[81];
    int InitiallySet;
    general_exception EventException;
    unsigned long EventWait;
public:
    event_mutex(char *MName, int Initial =0,
                unsigned long Dur=SEM_INDEFINITE_WAIT);
    void postEvent(void);
    void waitEvent(void);
    ~event_mutex(void);
};
```

postEvent()和 waitEvent()成员函数可以在生产者和消费者构建对象期间使用。消费者的构造函数可以使用 waitEvent(), 让对象直到从生产者接收到广播后才能完成。生产者将使用 postEvent() 来信号通知消费者，让消费者知道共享队列中已经存在可处理的内容，可以继续构建和消费了。前面讲过，postEvent()和 waitEvent()成员函数只是相应操作系统 API 的封装器：

```
void event_mutex::postEvent(void)
{
    DosPostEventSem(EventMutex);
}

void event_mutex::waitEvent(void)
{
    DosWaitEventSem(EventMutex, EventWait);
}
```

在这里，postEvent()和 waitEvent()函数为 OS/2 事件信号量提供封装器。图 11-5 显示了如何使用一种简单 Gantt 图来说明同步关系。

图 11-5 显示了若干对象构造函数间的同步关系。对象 A 与对象 C 之间存在 FS 关系。对象 D 与对象 B 之间存在 SS 关系。对象 E 与对象 A 之间存在 FF 关系。将用这些同步关系来描述不同线程中成员函数间的关系。它们提供了一种让程序员形象化多个线程中的对象是如何相关联的机制，这种机制与执行同步有关。同步关系也适用于任何类型的 C++类成员函数之间。

11.3.3 析构函数与 FF 关系

要小心识别不同线程中依赖对象间的 FF 关系。如果对象 A 位于另一个线程中，对象 B 位于

另一个线程中，对象 B 依赖对象 A，并通过 this 指针与对象 A 连接，那么，我们不希望在析构对象 B 之前析构对象 A。在这种情况下，对象 A 与对象 B 之间具有 FF（finish-to-finish）关系。如果对象 A 在对象 B 之前析构，则对象 B 很可能执行非法内存访问，导致某种形式的页错误或片断保护违规。在 threaded_calculator 服务器中，我们看到的就是这种关系。请看下面的代码：

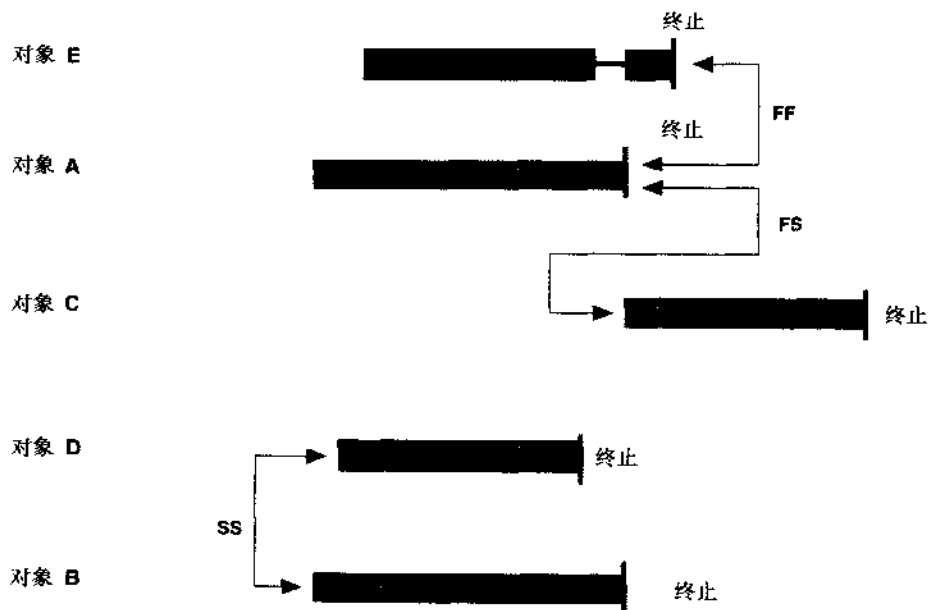


图 11-5 用一种 GANTT 图说明对象 A、B、C、D 和 E 的构造函数的同步关系

```
double threaded_calculator::evaluate(string Input)
{
    ct_thread Thread;
    double Temp;
    lock();
    InputString=Input;
    Thread.begin(::evaluate,this);
    Thread.wait();
    Temp=Result;
    unlock();
    return(Temp);
}
```

其中的 wait() 函数导致包含 evaluate() 成员函数的线程在 :: evaluate(this) 友元函数完成前一直阻塞。尽管 wait() 不处于析构函数中，但它执行的是同样的功能。

11.3.4 线程集合与对象

在使用接口类封装操作系统 API 时，重要的优势是封装（encapsulation）和保护（protection）。

在多线程环境中, 保护互斥量句柄或条件变量不漂浮不定是消除导致竞争条件、死锁以及无限延迟情况发生所必需的。我们还可以使用封装来控制线程取消, 即通过另一个线程的调用终止一个线程的执行。所需的所有调用线程为线程 ID 或即将取消线程的线程句柄。调用线程调用如下所示函数:

```
pthread_cancel(ThreadId)           //Posix Pthread
TerminateThread(ThreadId,ExitCode) //Win 32
DosKillThread(ThreadId);           //OS/2
```

取消线程可以有效停止它的跟踪和销毁。线程取消使得执行过程变得更复杂。一般情况下, 它带来的弊病多于益处。小心鉴别对线程取消的一些评论。举一个常见的例子, 有一系列线程, 它们都对大型数据库执行检索。一个线程在其它线程之前检索到了结果。因为让其它线程检索已经发现的数据是一种资源浪费, 所以程序员用一个致命的线程取消函数来取消它们。潜在的害处何在呢? 在某线程接收到取消命令时, 就不容易预测此线程正在做什么。它可能正在写入一个长记录, 或者临时资源的句柄。它可能正执行一个操作系统 API, 此 API 正完成一个明确的任务。如果线程从自由存储空间创建了内存, 但在删除此内存之前被取消了, 就必定会产生内存碎片。如果线程处于关闭某文件的过程中, 而刚好此时被取消, 将会发生什么? 占有互斥量的线程可能在接收到取消命令时没有释放互斥量。线程被取消了, 互斥量仍然被占有。所有等待此互斥量的线程可能无限期地等待。如果线程已经被取消, 它不能释放它的互斥量。而且, 执行取消命令后, 调用了对对象的析构函数吗? 线程取消通常用 `abort()` 函数调用来实现。

有时, 线程取消不可避免。由于某些原因, 一个线程被锁定, 而且不能对任何通信尝试作出反应, 这样的线程必须取消。陷入致命或无限循环的线程必须取消。不过, 在大部分情况下, 应当使用条件变量和事件互斥量来控制其活动不再需要的线程。例如, 在上面大型数据库检索例子中, 发现请求项的线程应该广播通知其它线程, 让它们知道该项已经找到, 其它线程应当进行结构化地关闭。虽然有许多技术可以执行结构化线程取消 (structured thread cancellation), 但这些技术容易出错, 而且随着多线程应用程序变得越来越大、越来越复杂, 实现这些技术越来越困难。事实上, 一个有用的应用程序总要扩展, 变大变复杂, 这是必须面对的事实! 对于有些线程取消技术, 在可能的情况下, 可以适当使用封装和异常处理来替换。防止杂乱线程取消的第一步是在某个对象中封装线程句柄。我们可以声明如下线程:

```
#include<pthread.h>
typedef void *(*FunctionPtr)(void*);

class ct_thread{
private:
    pthread_t ThreadId;
    pthread_attr_t *Attr;
public:
    ct_thread(void);
    ct_thread(pthread_attr_t *Attribute);
    ~ct_thread(void);
    void begin(FunctionPtr PFN, void *X);
    void wait(void);
```

```
pthread_t threadHandle(void);
pthread_t threaded(void);
void cancel(void);
};
```

线程类封装了 `ThreadId`，它是一个私有数据成员。它只能被类的成员函数访问。唯一取消这个对象占有的线程的途径是通过 `cancel` 成员函数。如果线程对象被它的宿主对象私有性地占有，只有该宿主对象才能取消此线程。我们不必担心程序员会无意中在 `cancel` 调用中使用线程的 `ThreadId`，因为 `ThreadId` 由 `ct_thread` 类控制。如果线程的句柄为全局变量，或者可以被多个线程访问，它就有可能被错误地取消。随着在应用程序中添加更多的模块和线程，这种犯错的可能性也增加。如果应用程序中使用一条线程取消策略，则错误的线程可能被取消。最好使用线程间通信来决定线程的起始和终止。在尽可能的地方执行合理结构化的线程关闭，而不是调用线程取消程序，这样更安全。如果必须线程取消程序，应当从占有此线程的对象中调用，而且制定适当的线程清理策略。

11.3.5 线程与异常处理

多线程编程时可能发生多种潜在的问题。大部分并行编程中存在的缺陷也存在于多线程编程中。这两种类型编程之间的区别不是结构上的显著差别，只是一种细微程度上的差别。在支持多线程处理、并发以及并行处理环境中常见的问题包括：

- 数据竞争；
- 死锁；
- 无限延迟；
- 优先权倒置。

在使用过程化方式编程的中型到大型系统中，这些问题难以跟踪和控制。我们演示了可以用于避免这些问题的面向对象编程技术。封装了互斥量和线程句柄，而且直接关联互斥量和线程与使用它们的类之后，我们就消除了多线程应用程序中可能出现的潜在缺陷。我们还可以做更多的工作，使用 C++ 的异常处理机制，让多线程应用程序更强健。C++ 异常处理机制不仅仅是一种错误处理工具。它提供了一种面向对象容错（fault tolerance）方式（Stroustrup, 1994）。即使系统组件失败时，容错系统也能继续操作。容错性既可应用于硬件，也可以应用于软件系统。在多线程应用程序中，我们对软件系统中的容错性感兴趣。虽然硬件可能是软件问题的起因，但我们希望着重讨论 C++ 异常处理机制的使用，帮助我们面临一定的软件失败时实现容错性。

异常处理所隐含的基本思想是让不能处理特定问题的组件将该问题传递给另一个知道如何解决此问题的组件。使用异常处理，我们可以设计特殊目的的组件，它的特定功能就是应付错误（在发生错误的情况下），并且处理其它软件异常。C++ 由 `try{}、catch{}以及 throw{}结构` 来支持异常处理。`try{}结构` 用于封装我们希望在面临软件问题时仍然能继续的系统部分。不知道如何处理特殊问题的组件使用 `throw{}结构` 将该问题传递给可以处理它的组件。知道如何处理特殊软件异常的组件使用 `catch{}结构`。`catch{}结构` 用来标识哪一个组件可以处理哪种类型的软件问题。

异常处理为 C++ 程序员提供了另一种处理错误、异常程序执行以及意外情形的方式。错误处理的传统技术包括设置全局错误标志、从函数返回错误值。两种技术通常在大型程序中失效，

甚至在多线程程序中，这两种技术带来的问题比提供的解决方案还多。使用传统方式来处理错误和异常难以实现容错性。在多线程应用程序中包含错误代码全局变量可能导致数据竞争问题。如果我们有二个同时访问某不安全库函数的线程，而且设置了一个全局错误标志，那么，到底是哪一个线程导致错误标志被设置呢？如果二个线程将错误变量设置成不同的值，我们该怎么办呢？在面向对象多线程应用程序中，若干重要的函数不会返回值。特别是，构造函数和析构函数没有返回值。因为我们经常在构造函数和析构函数中使用互斥量和条件变量，如果构造函数或析构函数碰到异常情况，我们该采取什么措施呢？我们可以使用 C++ 的异常处理能力来使多线程架构更可靠。

下面描述了 C++ 异常处理机制提供的 3 种重要功能：

- 允许没有返回值的函数抛出异常对象。
- 允许函数将抛出结构用作替换返回机制来返回多种类型返回值。
- 允许程序员以异常对象的形式定义情景敏感性诊断 (situation sensitive diagnostics)。这些异常对象可以让异常情景更容易理解、维护和调试。

我们可以在多线程应用程序中利用这些功能带来的直接好处。让我们考察互斥量接口类，作为一个异常处理如何与没有返回值的函数一起使用的例子。下面是简单互斥量接口类的声明：

```
#include <eclasses.h>
#include <pthread.h>

class mutex{
protected:
    pthread_mutex_t Mutex;
    general_exception SemException;
public:
    mutex(void);
    ~mutex(void);
    void lock(void);
    void unlock(void);
};
```

在 mutex 类中有二个函数，它们不支持返回值。这些函数是 mutex 的构造函数和析构函数。如果构造函数不能创建互斥量变量，将会发生什么？前面讲过，mutex 类只是一个封装类或接口类。它封装对操作系统或线程库 API 的调用。如果对其中一个 API 的调用失败时怎么办？因为构造函数没有返回值，所以没有切实的办法让 mutex 类的用户知道发生了问题。对于 mutex 类的析构函数同样如此。下面是 mutex 类的构造函数和析构函数的声明：

```
mutex::mutex(void)
{
    if(pthread_mutex_init(&Mutex, NULL)){
        SemException.message("Could Not Create Mutex");
        throw SemException;
    }
}
```

```

mutex::~~mutex(void)
{
    if(pthread_mutex_destroy(&Mutex)){
        SemException.message("Could Not Destroy Mutex");
        throw SemException;
    }
}

```

构造函数调用 `pthread_mutex_init()`。析构函数调用 `pthread_mutex_destroy()`。如果其中任何一个 API 失败，构造函数和析构函数可能抛出 `SemException`。所以，尽管构造函数和析构函数没有返回值，但可以抛出异常的能力允许互斥量对象通知用户发生了问题。`SemException` 是一个 `general_exception` 类型的对象。下面是 `general_exception` 类的声明：

```

class general_exception{
protected:
    char Operation[2];
    char Message[81];
public:
    general_exception(void);
    general_exception(char *Msg);
    general_exception(const general_exception &N);
    general_exception &Operator=(const general_exception &N);
    void operation(char *Op);
    char *operation(void);
    void message(char *Msg);
    char *message(void);
};

```

`general_exception` 类是一个常规 C++ 类。在这个实例中，它的主函数记录发生错误的描述。我们将 `SemException` 声明为一具 `general_exception` 类型的对象。如果由于某些原因 `mutex` 的构造函数或析构函数不能执行它的函数，它将传递适当的消息给 `SemException` 对象，然后使用 `throw()` 结构将 `SemException` 对象传递给另一个组件。让我们进一步考察这个过程。这个小程序声明了几个 `mt_rational` 对象。它创建这些对象，并希望对它们进行算术运算。因为这一过程包含在一个 `try{}` 块中，所以任何抛出异常都将传递给第一个合格的 `catch{}` 块：

```

#include <iostream.h>
#include "mtration.h"
void main(void)
{
    try{
        mt_rational X(3,0);
        mt_rational Y;
        mt_rational Z[2];
        Z[0].assign(1,2);
        Z[1].assign(5,6);
        Y.assign(6,4);
        cout<<X+Y+Z[1];
    }
}

```



```

    }
    catch(general_exception &X)
    {
        cout<<X.message()<<endl;
    }
}

```

图 11-6 显示了 `mt_rational` 类的类关系，这个类包含一个 `mutex` 类，而且从 `rational` 类继承了 `general_exception` 类。`mutex` 类还包含一个 `general_exception` 类。如果用户试图以 0 为分母构建一个 `mt_rational` 对象，就会抛出一个异常，因为分母为 0 的有理数无定义。

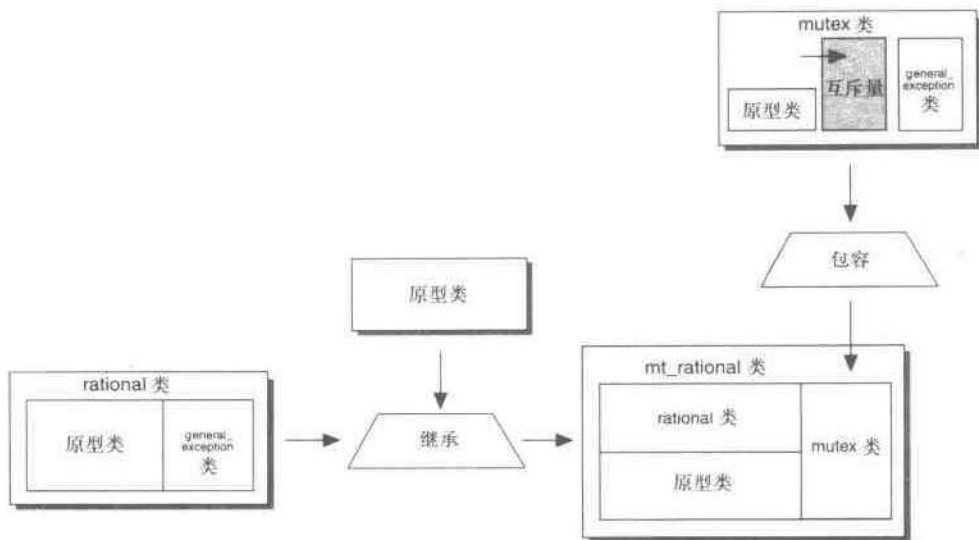


图 11-6 `mt_rational` 类的类关系图

下面是 `mt_rational` 类的构造函数：

```

mt_rational::mt_rational(long Num, long Den)
{
    if(Den==0){
        Exception.message("Zero Is A Invalid Denominator");
        throw Exception;
    }
    Numerator=Num;
    Denominator=Den;
}

```

而且，设计 `mt_rational` 类让它能用于多线程环境中。图 11-6 显示 `mt_rational` 类包含一个用于保护 `mt_rational` 类的临界区的 `mutex` 对象。但如果 `mutex` 类的构造函数失败了，情况又如何呢？构造函数将抛出一个 `general_exception` 对象。在 `try{} 块` 中，我们创建一些 `mt_rational` 对象：

```
try{
```

```

mt_rational X(3,0);
mt_rational Y;
mt_rational Z[2];
Z[0].assign(1,2);
Z[1].assign(5,6);
Y.assign(6,4);
cout<<X+Y+Z[1];
}

```

在第一种情况中抛出异常是因为我们试图用分母 0 来构建 X。这就创建了一个异常条件。流程控制在构建 X 时中断，将传递给最近的 catch{} 块，即：

```

catch(general_exception &X)
{
    cout<<X.message()<<endl;
}

```

catch{} 称做异常处理器 (exception handler) 或简称为处理器。它知道如何处理 general_exception 类型的异常。因为它带有一个指向 general_exception 的引用作为参数，它也可以捕获 general_exception 类型的所有后代。这种处理对象及其后代的能力是异常处理胜过传统错误处理的诸多优点之一。通过这一个特征，程序员可以设计异常类层次。图 11-7 显示了在线程处理环境中使用异常类层次的情形。

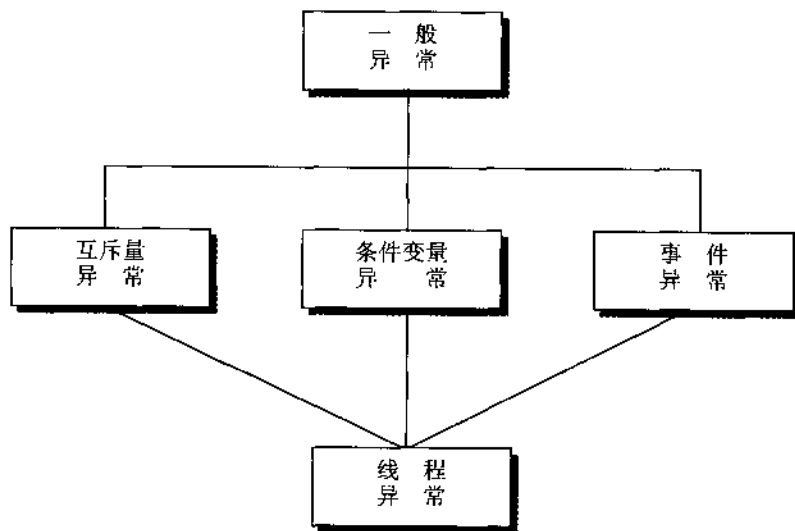


图 11-7 多线程环境中使用异常类层次的例子

每个从 general_exception 派生的类用于专门处理特定类型的异常。互斥量异常可以给它的处理器传递更正信息，这个处理器是针对潜在的竞争条件。条件异常可以通知它的处理器关于无限延迟情况的细节。处理器可能设计用来基于它所捕获的异常对象进行恢复。因为线程异常、互斥量异常、条件异常以及事件异常都派生于 general_exception，所以可以捕获 general_exception

类型对象的所有处理器都可以捕获线程异常、条件异常等类型的对象。对 `mt_rational` 对象执行算术运算这种情况, `catch{}` 块可以捕获 `mt_rational` 中的互斥量对象抛出的异常, 或者捕获 `mt_rational` 的构造函数抛出的异常。

一、调用堆栈和线程

异常处理的一般形式允许异常沿着函数的调用链向上传播。例如:

```
funcA()
{
    try{
        funcB()
    }
    catch(ErrorTypeA X)
    {
        X.doSomething();
    }
    catch(ErrorTypeB X)
    {
        X.doSomething();
    }
}

funcB()
{
    try(
        funcC()
        funcD()
    )
    catch(ErrorTypeB X)
    {
        X.doSomething();
    }
}

funcC()
{
    throw ErrorTypeB
}

funcD()
{
    throw ErrorTypeA
}
```

在这个调用序列中, `funcA()` 调用 `funcB()`, `funcB()` 调用 `funcC()` 和 `funcD()`。注意, `funcB()` 包含一个 `try{}` 和 `catch{}` 块。同样, `funcA()` 包含一个 `try{}` 和 `catch{}` 块。 `funcC()` 和 `funcD()` 抛出

不同类型的异常。虽然 `funcD()` 被 `funcB()` 调用, 但 `catch{}` 处理器并不处理 `ErrorTypeA` 类型对象; 它只处理 `ErrorTypeB` 类型对象。所以, 当 `funcD()` 抛出异常时, 它将沿着调用链传递, 直到到达第一个处理 `ErrorTypeA` 类型对象的处理器。在这种情况下, `funcA()` 中的处理器将最终捕获 `funcD()` 抛出的异常。请注意 `funcC()` 抛出的是 `ErrorTypeB` 类型异常。在处理 `ErrorTypeB` 异常的函数调用链或调用堆栈 (call stack) 中最近的处理器是 `funcB()` 中的处理器, 它将捕获这个异常并进行处理。不过, 如果 `funcB()` 没有 `ErrorTypeB` 类型对象的处理器, 异常将沿着调用堆栈传递, 最终被 `funcA()` 中的处理器进行处理。理解以上处理顺序很重要。如果某函数抛出一个异常, 该异常沿着调用链传递一级。如果这一级的函数没有处理此异常的处理器, 则异常沿着调用链传递到另一级, 直到找到合适的处理器为止。如果没有发现处理器, 程序员将退出。

从以上介绍可见, 异常的抛出和捕获是通过调用堆栈的协助来完成的。请记住, 进程中的每个线程都有自己的调用堆栈。因此, 不要试图从一个线程到另一个线程抛出异常, 例如:

```
main()
{
    try{
        pthread_create(&Tid, funcC, NULL);
        funcD();
    }
    catch(ErrorTypeA X)
    {
        X.doSomething();
    }
    catch(ErrorTypeB X)
    {
        X.doSomething();
    }
}
```

`funcD()` 和 `funcC()` 都抛出一个异常。`main()` 中的错误处理器将捕获 `funcD()` 抛出的 `ErrorTypeA` 对象, 因为 `funcD()` 是 `main()` 调用堆栈的一部分。我们可能这样认为, 因为调用了 `funcC()`, `main()` 的 `try{}` 块也位于 `main()` 的调用堆栈中。情况并非如此。一旦 `pthread_create()` 创建了另一个线程, 也就创建了第二个调用堆栈。`funcC()` 抛出的异常没有处理器。这将导致程序自动终止。每个线程应当负责自己的异常处理。虽然可以设计在线程间传递异常处理对象的巧妙办法, 但并不推荐这样做。在设计异常处理逻辑时, 请单独考虑每个线程的调用堆栈。

二、通过异常处理返回多种类型

C++ 中异常处理机制为错误处理添加的另一个重要特征是允许函数返回不同的类型。例如, 有一个函数 `currentDate()`:

```
Date currentDate(String Day);
```

这个函数以字符串形式接受一个日期, 并作为 `Date` 对象返回此日期。如果传递给 `currentDate()` 函数的字符串是有效的某公元中的某一天, 则用户将取回一个构建正确的 `Date` 对象。如果用户传递的字符中不包含有效日期, 例如是一个名字或另一个序列, 情况如何呢? `currentDate()` 应当返

问什么？我们通过 C++ 异常处理功能可以编写以下代码：

```
Date currentDate(String Day)
{
    Date Temp
    if(!validDate(Day){
        throw Exception;
    }
    TempDay(Day)
    return TempDay;
}
```

如果一切顺利，我们就可以取回一个 Date 对象；如果不顺利，抛出了异常，我们可以作出其它安排。从一个函数调用取回不同类型的能力只限制于两种类型。函数可能抛出任何类型的数字，这意味着异常处理可以用作表达软件异常的一种极其强大机制。

三、可靠线程化架构

在多线程应用程序中出现的许多情况都可以很好地应用异常处理。使用封装连接互斥量与它们应当保护的临界区。使用封装来保护条件变量句柄、事件句柄以及互斥量句柄。设计由适当成员函数自动完成锁定和取消锁定的类。虽然在消除多线程环境中发生的问题上我们已经迈出了很大的一步，但这并不能完全防止并发执行线程可能导致的意外失败。异常处理提供了另一道防线。作为异常抛出的对象携带着解释什么地方出错的诊断信息。它们还携带异常处理器可能包含的其它对象和动作。

例如，请看下面一种情形，线程 A 阻塞，一直等待着线程 B 占有的资源，同时线程 B 也阻塞，它等待着线程 A 占有的资源。如果对方不释放资源，两个线程取消阻塞，就会发生死锁。可以借助于异常处理机制来识别和处理死锁。抛出异常的进程可以包含标识死锁情形的逻辑。抛出对象可以包含针对每个侵犯线程的线程对象。处理器然后决定做什么。处理器可以强迫从两个线程移除资源（如果可能的话）。处理器可以决定取消两个线程，从整体上保持系统的执行。处理器处理完死锁后，处理器可以执行一些可选路径，这是为碰到死锁后预备的。使用异常机制和用户自定义，异常类可以提高多线程应用程序顺利执行而不崩溃的成功率。

进程中的每个线程都至少有一个异常处理器，它可以处理并发执行期间所产生的特定问题的异常。当构造函数和析构函数为使用多线程应用程序中的部分对象时，它们应当包含异常抛出逻辑。判断死锁的架构部分应当在碰到死锁时抛出异常。异常处理和面向对象编程技术使我们离可靠多线程架构的目标更近了一步。

11.4 线程安全函数

“当函数在某一时刻被多个线程调用，而且不要求任何施加于调用者部分的动作时，函数或函数集被认为是线程安全的(thread safe)或可以重复进入的(reentrant)”(Klieman, Shah, Smaalders, 1996)。当设计一个多线程应用程序时，程序员必须小心，确保在同一进程内被多个线程并发调用的函数是线程安全的。应用程序常常调用来自附加库的函数。在许多场合，这些函数不是线程安

全的。也就是说，函数或者包含静态变量、访问全局数据，或不能重复进入。如果某个函数包含静态变量，那这些变量在函数调用间保持它们的值。函数为了正确操作要依赖于这些静态变量。如果多个线程并发使用一个包含静态变量的函数，则存在一个竞争条件。而且，如果函数修改全局变量，就像修改 `errno` 的函数一样，那么访问此函数的多个线程可能都会修改全局变量的值。如果访问全局变量没有被函数同步化，则存在一个竞争条件。重复进入代码（`reentrant code`）不包含可修改静态数据，或者使用没有同步的全局变量。如果某代码块不能在使用时更改，就认为它是可重复进入（Deitel, 1990）。可重复进入代码可以由多个并发线程共享，而不会创建竞争条件。如果函数不是可重复进入，或者访问未保护全局变量，或者包含静态可修改变量，则函数就是线程安全的。

当前大部分 C++ 编译器提供了标准库的多线程版本。在可能应用于多线程环境的时候，程序员应当连接标准库的多线程版本。

11.5 多线程环境中的不安全函数

在所有场合使用函数的多线程版本是不可能的。有时，特殊函数的多线程版本不适用于给定的编译器或环境。在其它场合，函数的接口不能简单地成为线程安全（Klieman, Shah, Smaalders, 1996）。而且，程序员可能面临在环境中添加多个线程的问题，而这个环境中使用的函数只设计用于单线程环境。在这样的情况下，一般经验是，使用互斥量封装程序中的所有此类函数。这可能有一些复杂。假如有 3 个并发执行的线程。其中两个线程（线程 1 和线程 2）都并发访问 `funcA()`。已知 `funcA()` 不是线程安全的。第三个线程（线程 3）不访问 `funcA()`，而是访问 `funcB()`。线程 1 和线程 2 都不访问 `funcB()`。为了解决 `funcA()` 的问题，程序员可能倾向于通过线程 1 和线程 2 使用互斥量简单地封装对 `funcA()` 的访问，如下所示：

```

thread 1          thread 2          thread 3
{
    lock()
    funcA()
    unlock()
}
{
    lock()
    funcA()
    unlock()
}
{
    funcB()
}
```

这看起来好像解决了问题，一个时刻只有一个线程访问 `funcA()`。此时的问题在于线程 3，它也与线程 1 或线程 2 并发执行，它访问 `funcB()`。如果 `funcA()` 和 `funcB()` 来自于同一个不安全库，它们可能都修改一个全局变量或静态变量。虽然只有一个线程访问 `funcB()`，但 `funcA()` 和 `funcB()` 并发执行，而且都修改全局变量。从中可以看出，竞争条件的发生并不总是明显的。在此情形下，由于对 `funcB()` 的访问没有同步化，所以可能不知不觉地引入了竞争条件。如果不知道来自库的哪一个函数是安全的、哪一个是不安全的，程序员就有 3 种选择：

- 不要在应用程序中使用任何不安全函数。
- 限制在单线程中使用所有不安全函数。
- 通过一套同步机制封装所有的可能不安全函数。

虽然第三个选择似乎有一点极端，但如果打算在多个线程中使用不安全函数，这通常是必需

的。自然，笔者在这里提倡为所有即将用于多线程应用程序中的不安全函数提供接口类。一旦将不安全函数封装在接口类中，接口类或通过继承或通过复合可以与适当的同步对象结合在一起。然后通过继承或复合，接口类可以与它的宿主类结合，这样有效地消除了出现竞争条件的可能性。

11.6 在多线程架构中使用 STL 算法

C++标准库最新添加的内容之一是 STL (Standard Template Library, 标准模板库)，它是一套容器类以及可以应用于这些容器类的一般性算法。STL 可以简单划分为 3 个主要组件：

- 容器类 (container class)；
- 迭代器 (iterator)；
- 算法 (algorithm)。

图 11-8 显示了这 3 个基本组件之间的关系。迭代器充当容器和算法间的连接点。

迭代器访问容器，同时算法访问迭代器。3 个组件间的这种结构和关系相当强大和灵活。不过，这些组件间的结构以及它们的关系同时违反了许多面向对象编程的思想。违反的主要思想是面向对象编程的核心：即封装。“封装原则保证它的状态只能被它自己的过程来修改”(Cox, 1987)。STL 算法不是容器的成员函数。但是它们可以直接用于修改容器的状态。迭代器是许多编程问题的强大解决方案，它们也是可用于分离对象的函数与对象的数据的一般性指针。在面向对象编程中，其中的一个主要目标是以类的形式绑定函数与数据。迭代器可用于取消这种绑定。根据 Brad Cox 编著的 *Object-Oriented Programming—An Evolutionary Approach* 一书，他认为“封装是整个途径的基础。它的贡献是在每段数据周围放置一道代码墙，以此来限制修改带来的影响”。不仅仅是 STL 的基本结构对于纯面向对象导致问题，3 种基本 STL 组件间的关系也会在多线程环境中产生问题。肯定的是，当在多线程环境中使用容器对象时，迭代器操作会使容器对象的修改控制难以进行。

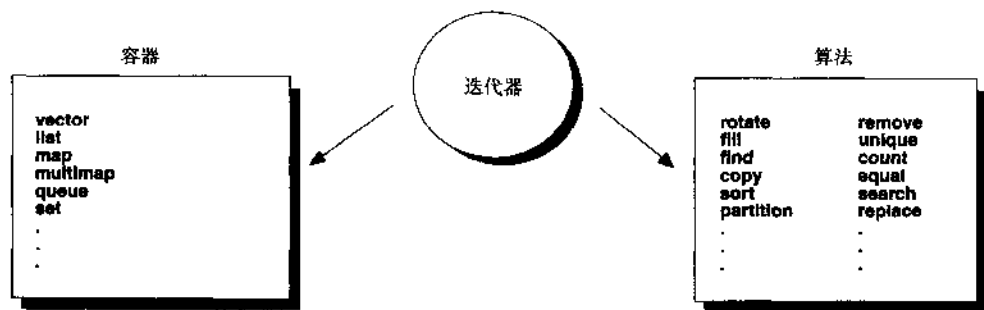


图 11-8 迭代器、容器类以及算法间的关系

我们的目标是避免自由漂浮的函数和数据。我们希望用函数封装数据。在多线程应用程序中，我们要使用封装的基本原则保护临界区。因为 STL 算法不属于任何特定类，所以它们只是自由漂浮的函数。而且，因为迭代器不是容器对象的直接成员，但是可以直接修改容器对象的数据，这就导致了问题的存在。如何在容器对象内保护临界区呢？如果我们为迭代器提供某种类型的锁定，

这不会解决此问题。迭代器可能被锁定，但容器对象的成员函数呢？锁定迭代器不会阻止容器对象被它自己的成员函数的修改。我们可以尝试为容器对象的成员函数提供内部锁定。我们将在容器的 `insert()`、`erase()` 和 `push_back()` 类型操作中放置互斥量锁定。这种途径的问题在于，迭代器仍然用于绕过锁定和修改容器对象内的临界区。我们可以尝试既锁定迭代器，也锁定容器对象。这听起来好像可以解决问题，但如何在迭代器和容器对象间提供公共互斥量和条件变量呢？要让迭代器一致，就要重新设计。不过，重新设计 STL 超出了本书的范围。

牺牲灵活性换取线程安全

我们还有另外一种方案，牺牲迭代器提供的灵活性，而换取容器对象和算法可以在多线程程序中安全使用。同样，我们使用接口类来解决问题。我们的策略是通过复合结合容器类与接口类，而且使用成员函数封装 STL 算法。接口类的用户只能通过接口类的成员函数访问算法和容器。

`lqueue` 类如下所示：

```
#include <deque.h>
#include "ctmutex.h"

template <class T> class lqueue:virtual private named_mutex,
    virtual private event_mutex{
protected:
    deque<T> SafeQueue;
public:
    lququ(char *MName,int Own,char *EName,int Initial,
        unsigned long Dur);
    inline void insert(T X);
    inline T remove(void);
    inline T front(void);
    inline T back(void);
    inline unsigned int empty(void);
    inline unsigned int size(void);
    inline void erase(void);
    inline void reversed(void);
    void wait(void);
    void broadCast(void);
};
```

它包含一个 `deque<T>` 对象，名字为 `SafeQueue`。`lqueue` 类是一个接口类。它的唯一目的是控制对 `deque<T>` 对象的访问。通过使用接口类，我们可以使对 `deque<T>` 对象的访问与多线程环境兼容。而且，防止类的用户在 `lqueue` 类的 `deque<T>` 组件中应用自由漂浮的 STL 算法。只有在 `lqueue` 类成员函数中封装的算法才能访问 `deque<T>` 组件。例如，`reversed()` 成员函数：

```
template <class T> void lqueue<T>::reversed(void)
{
    lock();
    reverse(SafeQueue.begin(),SafeQueue.end());
    unlock();
}
```


它使用了 STL `reverse()` 算法。`reverse()` 通常是一个自由漂浮函数。也就是说，它不是任何类的成员函数。它可以被正确定义了迭代器的类自由使用。不过，我们在这里封装了 `reverse()`。如果 `lqueue` 类的用户想倒置 `deque<T>` 组件中的数据项，用户必须使用接口类的 `reverse()` 成员函数。这也为线程化环境提供了自动锁定和取消锁定处理。

通过接口类和封装，我们将 STL 容器和迭代器调整为一种多线程程序可以接受的形式。通过准确理解线程中的同步关系和异常处理，我们可以预测对象在多线程环境中的行为。不过，要绝对准确地预测对象的行为，必须对它们进行全面地测试。我们将在下一章讲解在支持并发的环境中测试对象时所需要的步骤。

测试多线程应用程序

创造一个世界：赋予意图以形式、显现梦境、形象化未见事物……这是上帝的工作，不是吗？我们只是人类，但当我们开发这种技术并为个体和社会应用构建世界时，我们也承担了一定的责任。

Virtual Worlds
——Meredith Bricken

测试计算机软件的方法有多种，同时也存在多种类型的软件测试（software test）。在软件设计和实现阶段完成软件的测试。还有安装（installation）和操作上的测试（operational test）。软件开发者使用可用性测试来判断软件包的用户友好性。用可接受测试来判断客户是否对交付系统满意，同时测试软件测试的完整性。因为当今软件系统的复杂性，以及它们的经济和社会价值，所以软件测试成为一种高专业化的领域，它需要专业训练和专业工具。本章讨论在设计和实现用于多线程环境的面向对象系统时可以采用的一些基本软件测试类型。

12.1 软件测试的目标

以简单的眼光看软件测试，会认为软件段的测试目标是判断软件能否正常工作。以后我们会看到，这仅仅是软件测试的多个目标之一。软件测试核心工作围绕发现和防止软件缺陷与软件故障（software failure）为中心。根据 Musa、Iannino 和 Okumoto 的观点，软件故障是“一种缺陷、缺失或多余指令或相关指令集，它们是生产潜在或实际故障类型的原因”。而且他们将软件故障定义为“程序操作客观结果偏离程序对运行的要求”（Musa、Iannino 和 Okumoto, 1987）。换句话说，软件故障是软件段中逻辑或数据表示上的缺陷。软件故障是执行包含缺陷程序代码的结果。

最常见形式的软件故障或缺陷在软件开发的编码、编译和崩溃循环中发现，这个循环过程是每个程序员都已经熟悉的。不幸的是，编码、编译和崩溃循环已经成为开发许多商业、工业，甚

至科学软件系统的一种事实上的方法。在编码、编译和崩溃循环中，软件开发者编写一段代码，执行这段代码，但因为某些软件缺陷（漏洞）而导致代码崩溃。开发者使用调试器，或其它一些方法来查找缺陷。开发者接着排除缺陷（漏洞）并再次启动编码、编译和崩溃循环。在开发者排除了所有导致程序停止的漏洞后，这个过程才算结束。软件有幸完成合理的输入并且产生合理的输出后，该系统就被认为通过了测试！但是，如果软件缺陷没有被开发者或测试者发现，这些软件缺陷可能在用户执行程序时变成故障。

软件测试的主要目标是通过辨别和排除软件缺陷，或者防止软件缺陷，防止软件故障的发生。问题是，我们如何发现或防止软件缺陷呢？这个问题对于面向对象系统特别难以回答。对象的设计者常常完全不知道面向对象组件的使用方式，或它使用的上下文。在对象设计中使用多态能够让集成测试难以进行。类的设计者可以把成员函数声明为纯虚函数，指定这个虚成员函数必须实现。不过，类的设计者不能确保在派生类中如何实现这个纯虚成员函数。如果我们使用复合或继承来构建对象，我们用来构建对象的源对象必须是可靠的。通常设计某个对象作为一些未定义、不可知的未来对象类型的基础。如果我们不知道如何或在哪里使用面向对象组件，我们如何才能找出或防止潜在的软件故障呢？测试面向对象和多线程软件没有现存的合适办法。但是，在测试面向对象软件时，应该遵守几条重要的原则：

- 在软件系统还未进一步复杂化前及时测试。
- 将软件遇到的缺陷类型分门别类。

12.1.1 分而治之（divide and conquer）

我们曾经建议程序员使用增量多线程处理技术在应用程序中引入多线程技术。我们也建议对软件系统进行增量测试。如果可能的话，用最简单的形式测试软件，测试每个组件，然后测试组件组。测试单个组件属于所谓的单元测试（unit testing）。测试交互的组件组属于所谓的集成测试（integration testing）。我们将在后面讨论这两种测试。

在程序员开始排除软件段中的缺陷前，他应当知道在这个软件段中可能存在的缺陷类型。程序员应该对软件段中可能存在的每一种缺陷制定一个测试计划。也就是说，程序员要对每一类软件缺陷进行测试。表 12-1 列出了针对包含多线程处理、并发、并行或同步进程程序的一些典型软件缺陷类别。

表 12-1 包含多线程处理、并发、并行或同步进程程序中的一些典型软件缺陷类别

缺陷类别	描 述
竞争条件	发生在两个或更多线程或进程试图同时修改相同共享可修改数据块之时
死锁	线程或进程等待某件永远也不会发生的事件
优先权倒置	当较低优先权线程阻塞较高优先权线程的执行，同步变量被使用或竞争资源时发生
性能下降	当系统在反应灵敏性、执行时间长短、计算结果准确性等方面的性能下降时发生
无限延迟	当系统无限推迟进程或线程的规划，同时其它进程或线程要接收通知和资源分配，此时发生无限延迟

续表

缺陷类别	描 述
互斥量耗尽	当系统达到可以创建的最大数量互斥量时发生
线程耗尽	当系统达到可以创建的最大数量线程时发生

只有对合适的软件缺陷进行系统地分类才能保证测试者找到对应的缺陷。软件错误或缺陷有多种分类法。每种应用程序都在标准类别的基础上还另外添加了自己独特的类别。Cem Kaner 归纳了 13 种需要排除的一般错误类别 (Kaner, 1987)。这 13 种一般错误类别及其描述如表 12-2 所示。

表 12-2

13 种一般错误类别及其描述

错误类别	描 述
用户接口错误	设计、功能性或用户接口性能上的错误。用户接口可能在通信、命令结构或输出方面失败。程序可能不能完成用户希望完成的任务, 或者操作性能低下
边界相关错误	程序的边界是使程序更改其行为的分界线。不正确描述边界条件或者没有发现程序的极限可能会发生错误
计算错误	在计算中发生的错误。包括公式误解或丢失精度的错误。也包括由于使用不正确算法导致的计算性错误
初始和随后状态	在初始使用程序期间发生的错误。在程序每次重新启动时发生
竞争条件	当线程或进程在所希望线程或进程之前执行时发生
控制流错误	当程序错误的随后事情时发生此错误
错误处理	处理错误时发生的错误。程序可能没有发现或预料到可能的错误, 因而不能合理地纠正这些错误
处理或解释数据时的错误	在模块、程序、线程或进程相互间传递数据时, 误解、破坏或丢失数据时发生的错误
负荷错误	当程序过载时发生的错误。当程序长时期执行大量任务, 或者一次执行大量任务时, 可能会行为不正确。当程序耗尽内存, 或者与其它程序或函数共享资源时可能失败。程序满足极限的性能如何? 当超过极限时, 程序的性能又将如何
硬件	当程序没有识别硬件故障或从硬件故障中恢复时发生此错误。程序可能不能识别从设备返回的错误码, 或者试图访问一个不存在或已经在使用的设备。程序也可能给设备发送破坏的数据
源代码和版本控制	当使用不正确程序版本时发生该错误。包含错误的旧子程序版本可能连接到最新程序版本上
文档	当使用破坏文档时发生该错误
测试错误	在测试软件期间发生该错误

无论程序员将错误分成以上几种类别，还是决定使用其它更具体、更适合的类别，要记住的重要一件事是，要系统地标识软件组件中可能存在的软件缺陷类别。依赖于特别的编码、编译和崩溃这种软件开发循环方法并不能得到可靠的容错性系统。仔细检查软件，找出每种相关软件缺陷类别绝对是必需的。标识出执行于系统的测试类型，并设计测试计划，对每一种测试类型进行实例测试。

12.1.2 软件测试类型

标识软件缺陷类别后，必须设计处理每种缺陷类别的测试计划与测试实例。在每种类别中可以进行多种不同类型的测试，而且每种类别可以分成多种不同类型的测试。必须制定一个测试计划。测试计划实质是如何系统评估每种缺陷类别、使用不同测试类型的策略。表 12-3 列出了 7 种常见的测试类型；如果软件组件成功通过这 7 种测试，我们就得到了一个具有容错性的组件。

表 12-3 7 种常见的测试类型

测试类型	描 述
单元测试	要求一次测试软件的一个组件或单元。单元可以是软件模块、模块集合、函数、过程、对象、算法，有时也可以是计算机程序
强度测试	设计用来将某组件或系统推至极限，有时甚至超过极限。强度测试包括边界条件。当测试边界条件时，它帮助判断软件组件或系统处于边界时发生的情况
集成测试	用于测试组件的装配。将组件组合成逻辑组，每个组作为一个单元进行测试。这些组要接受单元所接受的同样类型测试。每当给装配添加一个组件，必须测试的元素数量相应组合性增加
递归测试	用于测试修改的模块。递归测试确保对组件的修改不会导致它丢失任何功能
操作性测试	用于测试系统的完整操作。这个测试将软件组件放在一个真实的环境中，加载全部系统负荷进行测试。组件在单元、集成以及强度测试间接受的测试通常也要作为操作性测试。操作性测试也用于判断组件在一个完全不同环境中的行为优劣
规范测试	用作软件验证过程的一部分。用原始规范审查组件。规范指定系统中要包含哪些组件，这些组件间应该存在什么样的关系
可接受测试	模块、组件或系统的最终用户用于判断其性能。可接受测试是软件确认过程的一部分

一、软件验证确认

对每种缺陷类别执行必需的测试类型，或通过更适合的测试类型检查每种缺陷类别。测试类型有两个基本目标：

- 软件验证 (software verification)；
- 软件确认 (software validation)。

有时，这两个目标可以表述为以下两个问题：

- 我们正确构建了此软件吗？
- 我们构建了正确的软件吗？

当我们从事软件验证时，我们实际上是回答第一个问题。验证是利用软件规范对软件功能的一种审查。软件规范产生于设计阶段。当执行软件验证时，我们检查软件是否满足这些规范。当我们判断软件是否实际执行着用户希望的任务时，我们实际上在回答第二个问题。验证是正确实现软件，确认是实现正确的软件。大部分软件测试过程可以描述为验证或确认过程。最后，要回答我们是否正确地构建了正确的软件这一问题，必须执行多种不同类型的测试。

二、单元测试

单元测试 (unit testing) 要求对软件进行一次一个组件地测试。根据软件架构的不同，单元也不相同。单元可以是一个软件方法、一个模块集合、一个函数、一个过程、一个对象、一个算法，有时也可以是一个计算机程序。在面向对象软件结构中，单元则是一个对象，或至多是一个对象层次。单元测试回答验证方面的问题。也就是说，组件是否满足规范？除非组件被认为是最终产品，否则单元测试就不能回答验证问题。在单元测试中，测试组件的最基础部分。如果我们对一个对象执行单元测试，单元测试可能分散于该对象的几个主要方面：

- 对象的组件复合；
- 成员函数访问数据组件；
- 成员函数正确性；
- 对象的过渡状态；
- 成员函数调用序列；
- 对象完整性。

12.1.3 对象的组件复合

如果对象通过继承或复合包含其它对象，在完成单元测试前，我们必须确保这些对象是可靠的。有时，所包含对象被证明是通过了测试的对象，在假定这些对象是可靠的情况下继续进行下一步。例如，如果我们声明 `MyObject` 是类 `A` 类型的对象：

```
class A{
protected:
    set<int, less<int>>>X;
    ...
    ...
    ...
}
```

则 `MyObject` 包含一个 `set<int, less<int>>>` 对象。这个 `set` 类是标准模板库的一部分，这个标准模板库现在是 C++ ANSI/ISO 标准的一部分。面向对象编程的一个主要优势在于代码重用 (code reuse)。对象成为标准库的一部分，我们可以基于这个对象进行构建，而不必重复测试这个对象的每个方面。除非由于编译器开发商的原因，在对象的实现中出现了问题，否则对象应当是可靠的。库对象应当只需要极少的单元测试！对于对象是标准 C++ 库的一部分的情况，我们可以假定这

些对象是可靠的而继续下一步。另一方面，如果我们通过继承或复合包含对象，它们是没有经过时间检验的对象，对此对象必须经过单元测试。为了测试对象的组件复合，必须测试包含在此对象中的对象。

12.1.4 成员函数访问数据组件

单元测试应当包括这样的测试：确保允许对象用户访问对象的任何数据组件所提供的成员函数不要作为私有、特定实现的数据。这个测试确保不遗漏任何成员函数。对于拥有多个成员函数的大对象，常常会忘记允许对一定数据成员的访问。而且，一些应当为对象私有接口一部分的成员函数可能会声明为公有。一些应当为对象受保护接口一部分的成员函数可能会声明为私有。对象的数据组件必须适当划分为公有、私有以及受保护部分。测试对象的数据组件判断对对象数据组件的访问是否正确进行了划分。在单元测试期间测试数据组件的访问。

12.1.5 成员函数正确性

对象成员函数不应该与常规函数混淆。对象成员函数可能极其简单，也可能相当复杂。对象成员函数可以只包含一条语句。另一方面，对象成员函数可能由包含多个算法和函数调用的复杂逻辑组成。对象成员函数为对象的用户提供服务。这些服务可能非常复杂。在一些复杂的对象中，一个服务可能包含多个程序。所以，测试成员函数的正确性与测试独立的程序一样具有挑战性。在单元测试期间，必须回答有关成员函数正确性的一些基本问题：

- 如果成员函数执行计算，计算正确吗？
- 成员函数是如何处理正确输入数据类型的？
- 成员函数是如何处理不正确数据类型的？
- 对于成员函数中包含的每个决策树将发生什么？
- 成员函数的逻辑是否包含正确的交易规则、断言、前置条件以及后置条件？
- 该成员函数访问临界区吗？
- 成员函数中的临界区是否受同步机制的保护？
- 组件的输出正确吗？
- 如果成员函数执行转换，输入正确的情况下，转换结果正确吗？
- 成员函数的执行是否有效，至少是否可接受？
- 成员函数的接口正确吗？它是否具备正确的访问策略？
- 是否编制了成员函数说明文档？

在单元测试期间应该回答关于成员函数的以上几个问题。当然，针对每个对象的具体域还必须回答其它一些问题。单元测试应用一种增量测试策略。每个组件单独测试。为了回答单元测试的问题，必须制定一个测试计划（test plan）和一套测试实例（test case）。测试计划是一种用于测试软件段的策略。测试实例是用于实现测试计划的数据和场景集合。单元测试判断软件组件在适当的条件下是否行为准确。在其它条件下测试软件的行为被看作是强度测试（stress testing）的一部分。必须注意，不同测试类型间的分界线常常不是太分明。强度测试可以在单元测试期间进行，它也可以作为集成测试的一部分进行等。

12.1.6 对象的过渡状态

对象的过渡状态表示对象可能所处的每一种状态。例如，某个对象可能被部分构建。在多线程或并发应用程序中，对象可能处于阻塞或挂起状态。在支持虚拟存储的环境中，可能将对象交换到磁盘外。对象可能处于析构状态、僵化状态等等。对象可能处于的每种状态必须在单元测试时进行评价。可以使用状态图来描述对象可能处于的状态。例如，假如有一个线程对象，我们可以用状态图显示线程对象的状态。图 12-1 是一个简单线程对象的状态图。

对象的数据组件将指明对象所处的状态。如果对象的数据组件部分被实例化，对象的行为将如何呢？在状态过渡评估时，必须评估每个对象的构造函数与析构函数。如果对象是一个应用框架，它是具备缺省行为？如果不正确地覆盖了虚函数，对象可能处于什么状态？

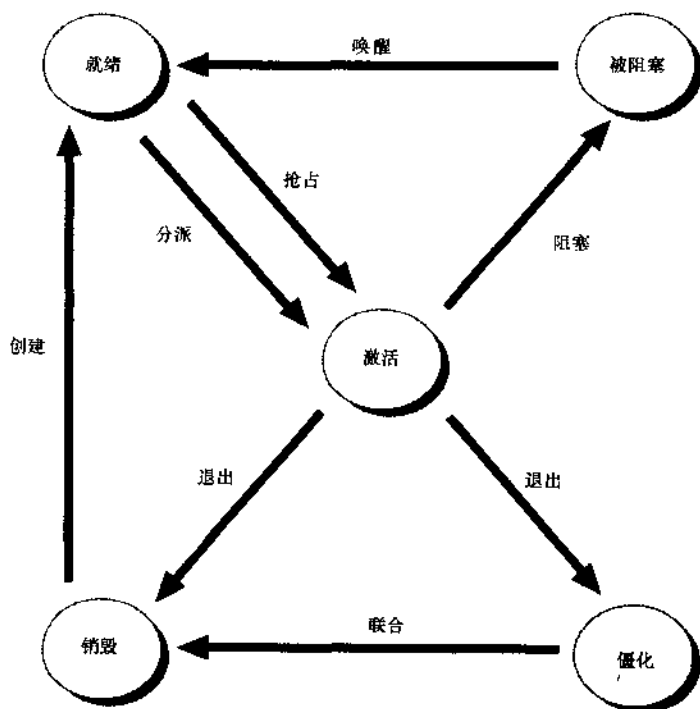


图 12-1 一个简单线程对象的状态图

如果对象模型是一个设备，对每种真实的设备状态是否存在一个对应的对象状态？如果对象占有资源，当对象抛出一个异常时，这些资源将处于什么样的状态中？

12.1.7 成员函数调用序列

对象为类的用户提供了一个服务列表。不存在指定的服务调用序列，除非对象是一个应用框架。Bertrand Meyer 在他编写的 *Object-Oriented Software Construction* 一书中称之为“购物清单方式”。“类是抽象数据类型可以使用的服务仓库（输出特征）...不存在相对重要性或相对优势的概

念：对调用服务的顺序不存在任何限制”（Meyer, 1988）。

当测试成员函数调用序列时，程序员按不同的顺序调用成员函数。如果我们要查看程序中某对象的成员函数的第二个调用的话，若对象有 15 个成员函数，成员函数的调用就有 15 种可能的选择。如果我们查看下面两个成员函数调用序列，就有 15^2 种成员函数调用的可能性。虽然不可能测试成员函数可能涉及的每种数量的序列，但至少必须确保在不锁定中崩溃系统的情况下，可以按任何顺序调用成员函数。比如对如下流对象：

```
class stream{
    FILE *Fp;
public
    stream(void);
    open(char *);
    read();
    close();
};

void main(void)                void main(void)
{                                {
    stream X;                    stream X;
    X.read();                    X.close();
    X.close();                  X.open("file");
    X.open("file");             X.read();
}                                }
```

同时还声明一个对象 X，然后在用不同的顺序调用这些成员函数，对象应当返回正确的值，而且不会由于成员函数按非法序列调用而导致对象的状态被毁坏。成员函数在执行它们的服务前应当检查对象的状态或数据组件。假定成员函数按某种人为的顺序调用是一个常见的设计错误。当测试成员调用序列时，如果成员函数按不同的序列进行调用，单元测试必须考虑对象的行为。

12.1.8 对象完整性

单元测试应该确保对象是完整的。这意味着对象包含所需要的数据成员和成员函数。如果目标是得到适应性强的对象，那对象应该具有极少的成员函数集（Carroll, Ellis, 1995）。在许多场合需要复制构造函数和对象赋值运算符。这些成员函数需要声明为 `const` 或保持可变性？如果在派生类中存在基类多个拷贝的可能性，基类要声明为 `virtual` 吗？这个类是否具有一个析构函数或多个析构函数？析构函数在必须声明为虚拟的地方是否声明为 `virtual`？对象是否具有异常处理逻辑和策略？需要用构造函数来解决所有必需的对象初始化和数据成员实例化吗？单元测试也用于判断对象组件是否完整。

一、强度测试

强度测试有多种形式和规模。强度测试设计用来将组件或系统推至其极限，有时甚至越过其极限。包含测试边界条件作为一种强度测试类型。当测试边界条件时，我们测试软件在软件组件或系统的逻辑边界处发生什么。例如，如果软件组件设计工作于多处理器或多线程环境中，当只

存在一个线程或处理器时将发生什么呢？组件如何执行？如果我们添加大量并行处理将会发生什么？也就是说，如果在目标环境中存在成百甚至上千个处理器时，情况将如何？组件的性能是否会受到最少数量线程或最多数量处理器的不利影响？

极端输入和输出

如果我们将输入推至逻辑的极限，看看软件组件会发生什么，这也是强度测试的一部分。如果组件执行一些数值计算，这些组件执行极其庞大或极其微小值的准确性如何？如果组件应该接收大写字母，而组件获得的是小写字母，将会发生什么呢？如果我们在某台计算机上执行一个组件，而这台计算机比我们最先设计该组件所用的计算机快上千倍时，情况有何不同呢？我们可能针对 640×480 分辨率的显示器来设计某个组件。如果该组件用于一台 1000×1000 分辨率的显示器上时，情况将发生什么变化？如果组件负责为显示设备或硬拷贝设备格式化输出，对于极其庞大或极其微小的值，数值格式化处理的效果如何？

资源耗尽

每个组件都需要一些类型的计算机资源。这些资源可以是处理器时间、外部存储器、内部存储器，也可以是计算机周边设备。如果可用资源比所需要的资源少时，将会发生什么？组件的行为如何？如果实际资源比所要求资源少，它是否会导致灾难性系统故障？有些系统可以在具有 64MB RAM 的环境中正确执行，而在具有 128MB RAM 的环境中，却反而奇怪地崩溃。如果组件设计用于连接有单个 CD-ROM 的计算机上，如果目标计算机上连接有多个 CD-ROM，将会发生什么呢？

多线程环境中的组件可能需要动态数量的互斥量。如果操作系统已经耗尽了所有可用的命名互斥量，组件将如何执行？组件是否抛出异常，或下降到单线程执行？如果组件是一个多线程服务器，而且为每一个客户创建一个新线程，如果系统用完了所有的线程空间，组件将会发生什么？大部分系统对于可以创建的线程有一个极限值。如果所请求的数量比系统可以产生的数量多，此时将会发生什么情况？整个服务器会被锁定，或者只是忽略最后一个客户？如果系统组件需要 200 个文件句柄，而系统只有 199 个，此时情况将如何？组件将失败，或组件的行为被调整？审查软件在资源耗尽方面的缺陷也归于强度测试之列。

资源耗尽对于依赖于动态数量线程、同步变量或处理器的多线程应用程序来说，这是一个严重的威胁。因为系统对于这些资源有一个极限，所以设计用于多线程环境的组件必须接受这方面的严格缺陷检查。

在强度测试期间，还必须进行的组件另一方面测试是，当存在严重阻塞时，组件将如何执行。如果组件需要执行 30 个线程，如果它所运行的环境中已经执行着 100 个其它线程，该组件将如何执行？面临严重系统阻塞时，组件的性能是否会下降得不可接受？相反，该组件是否假定已经执行有多个线程？如果发生的系统活动比预期的少，组件会崩溃吗？系统负荷测试是强度测试的一种形式。资源耗尽以及它的相关联测试资源过剩(resource surplus)在强度测试中一般要进行测试。

二、集成测试

集成测试(integration testing)的目的是测试组件的装配性。单元测试检查单个组件。完成单元测试和强度测试后，一次性将组件装配在一个组件中。然后将每个装配看作一个单元进行测试。

测试过程可以具有重复性和递归性。在集成测试中，组件按逻辑分级结合，每个组作为一个单元进行测试。这个组要接受的测试类型与单元接受的测试类型一样。每次对装配添加新组件，需要进行测试的元素数量要组合性地增加，因为新装配中接口间的可能交互数量以及决策树的数量和规模引入了组合性增长的选项需要测试。例如某对象具有 5 个成员函数。如果 3 个线程并发访问这个对象，访问对象时就有 5^3 种可能的成员函数组合调用。也就是说，第一个线程有 5 种可以调用的成员函数。第二个线程与第三个线程都有 5 种可能。因为我们不知道在某一时刻线程可能调用哪一个成员函数，所以集成测试必须考虑 5^3 种可能的组合。在集成测试时，对象的结构是区别它的一个方面。因为对象提供了无序服务列表，可能成员函数调用的组合性爆炸增长使得对象难以进行测试。例如，对于 `query_processor` 类：

```
typedef set<<char, less<char>> Vset;

class query_processor{
protected:
    set<char, less<char>> ValidOperands;
    set<char, less<char>> ValidOperators;
    set<string, less<string>> ValidFunctions;
    set<char, less<char>> ValidCharacters;
    set<char, less<char>> ValidLetters;
    set<char, less<char>> ValidNumbers;
    string InputString;
    list<expression_component> ExpressionTokens;
    stack<vector<expression_component>> OperandStack;
    stack<vector<expression_component>> OperatorStack;
    int ValidExpression;
    char FunctionResult[10];
public:
    query_processor(string Input);
    query_processor(string Input, Vset VOperands,
                    Vset VOperators,
                    Vset VFunctions,
                    Vset VCharacters,
                    Vset VLetters,
                    Vset VNumbers);

    query_processor(void);
    int checkParenthesis(void);
    int checkOperators(void);
    int checkDecimals(void);
    int checkCharacters(void);
    virtual void evaluate(void)=0;
    int parse(void);
    string inputString(void);
    void inputString(string Input);
    int validExpression(void);
```

```

double processNumber(int &Count);
char *processFunction(int &Count);
list<expression_component> expressionToken(void);
void process(void);
int validate(void);
);

```

打算将这个类用作一个应用框架类。为了对这个类执行集成测试，我们必须首先从它派生出另一个类。如果我们的目标是与另一个类一起使用这个类（比如 `calculator` 类），那么我们从类 `query_processor` 派生出 `calculator` 类，并作为一个单元测试它们：

```

class calculator:public query_processor{
protected:
    double Result;
    double processOperator(expression_component Operation);
public:
    calculator(string Input);
    calculator(string Input, Vset VOperands,
                                   Vset VOperators,
                                   Vset VFunctions,
                                   Vset VCharacters,
                                   Vset VLetters,
                                   Vset VNumbers);

    calculator(void);
    double result(void);
    void processCloseParenthesis(void);
    void evaluate(void);
};

```

请注意，`query_processor` 类声明为：

```
virtual void evaluate(void)=0;
```

`query_processor` 类与 `calculator` 类中成员函数组合的测试量相当艰巨。我们只希望派生类的实现者对 `evaluate()` 成员函数实现合理的解释。这个成员函数是 `query_processor` 类中的纯虚函数。这使 `query_processor` 成为一个抽象基类。如果我们不能准确知道这个 `calculator` 类如何实现 `evaluate()` 函数，如何才能测试这个基类呢？在这种情况下，我们必须依赖于文档和成员函数上下文与类的用户通信，知道 `evaluate()` 函数的打算应用的语义。因为对象一般是以数据为中心的，所以集成测试可能难以设计。

三、回归测试

回归测试（regression test）用于重新测试已经修改的模块。在类和对象中，有几种类型的修改自动要求进行回归测试：

- 对象的数据组件表示法改变时；
- 类中增添了数据成员时；
- 从类中删除数据成员时；
- 成员函数处理过程改变时；

- 添加了新成员函数时;
- 对象经过了继承处理;
- 删除了成员函数;
- 对象应用于新环境;
- 成员函数的声明改变, 例如改为虚拟函数;
- 给对象添加了额外的线程;
- 给对象添加了额外的临界区。

回归测试确保对对象的修改不会导致任何功能的丢失。除非对象的修改大幅度地更改了原始用法, 否则对象应当能够通过原始单元和强度测试。这是保存测试实例、测试数据以及测试结果的一个主要原因。对象经受修改时, 它应当接受原来的一套测试来判断修改是否破坏了对对象。单一回归测试并不足以测试对象的修改。与回归测试一起, 还需要新单元测试、强度测试以及集成测试。尽管这些测试既花费时间, 又耗费资源, 但这是必需的。

必须从两个角度来考察修改后的对象或类: 作为一种新对象与作为一种旧对象。它必须作为一种新对象来考察, 因为修改可能添加了新类或成员函数, 赋予对象额外的功能。回归测试没有涉及到添加的新功能。同时必须从旧对象的角度来考察, 是因为其它类层次可能依赖于原始的功能。原始设计对象的环境依赖于原始的功能。理想情况下, 对象不应该破坏现有的代码或环境。回归测试至少包含两套测试: 一套单元、集成以及强度的原始测试, 以及一套新的单元集成和强度测试, 后面这些测试用来验证对象的新功能。

回归测试似乎是多余的。不过, 通常对类的修改类型可以引入新的、难以觉察的漏洞类型。例如, 对象的内部表示是受保护的, 而且在许多情况下是私有的。对象的内部表示应该在不影响对象的公有接口的情况下能够修改。因此, 我们可能认为修改对象的内部表示不需要执行回归测试, 因为没有以任何方式更改公有接口。许多情况下, 这种假设是不正确的。如果我们更改对象的内部表示, 从矢量更改到树表示法, 我们就可能带来了潜在的资源耗尽问题, 而这个问题在旧对象中是不存在的。矢量可能是一个静态结构。树是一个动态结构。虽然对象的公有接口没有改变, 但对象如何使用系统资源却发生了变化。如果对象原来只要求 4MB 资源来构建或析构, 现在需要的资源大小可能改变了, 缩减至少于 4MB 或增加至多于 4MB, 这种变化可能破坏现有的代码。设计回归测试来捕获这种类型的软件缺陷。

四、操作性测试

操作性测试用于全面地测试系统的操作。这个测试将软件放到一个真实的环境中, 在完全系统负荷下测试。因为面向对象开发工作的结果可能只是一个对象或一个简单的对象层次, 因此操作性测试并不一定总是实用。在单元测试、集成测试以及强度测试中组件经受的测试也经常当成操作性测试。不过, 有两种类型的对象层次可以受益于操作性测试: 应用框架 (application framework) 与模式类 (pattern class)。这两种类型可以表示完整的应用程序, 在某些情况下甚至是系统。在真实的环境中实例化模式或应用框架, 看看它是如何实际执行的, 这是一种有意义的测试。如果模式类或应用框架将要应用于多线程环境中, 这种测试的意义尤其重要。资源耗尽、资源过剩以及系统性能是对应用框架或模式进行操作性测试的主要测试项目。正确性和功能性问

题已经在前面的测试阶段中得到了解决。操作性测试还用来判断对象在一种完全不同环境中的行为方式。也就是说,如果它以一种没有意料到的方式派生出子类,应用框架的行为将如何呢?如果它以一种通常情况下不会组合的模式来使用,模式类的行为将如何呢?操作性测试提供了一种最终类型的强度测试。

五、规范测试

规范测试(specification testing)是软件验证过程的一部分。这种测试是用原始规范来审查对象。面向对象系统分析阶段和后续面向对象系统需求阶段将产生一套对象规范。这些规范指定系统中要包含什么样的对象,这些对象间存在什么样的关系。这些规范决定了成员函数接口协议。规范提供了对象层次、模式以及复合。规范测试将判断对象或对象组是否提供了所要求的功能。显然,规范测试在单元测试和强度测试前完成。我们不希望对那些并不满足原始规范的对象执行耗时耗力的单元或强度测试。对不满足原始规范的对象执行广泛的测试是一种资源浪费。但是,我们可以执行一些预备性的集成测试来检查对象是如何结合在一起的。一些集成测试通常是判断对象是否满足规范所必需的。

六、可接受测试

可接受测试(acceptance testing)是由模块、组件或系统的最终用户执行的测试。对于对象,结果可能是一个由其它程序员或软件开发者使用的类库。事实上,对象或对象类层次的最最终用户通常是软件开发者。虽然我们前面提到的测试是由对象的生产者执行的,但可接受测试是由消费者或对象的用户执行的。可接受测试是软件确认过程的一部分。验证过程判断我们是否正确构建了软件,确认过程判断我们是否构建了正确的软件。可接受测试对于对象的生产者难度相当大。

因为对象在继承、覆盖虚成员函数、引用基类、引用派生类以及定义纯虚成员函数的过程中可能会发生变化,所以,对象的开发者不能预测对象在可接受测试中的所有使用方式。事实上,由于目标对象通常是未知的,所以可能在可接受测试中导致奇怪的结果。在这种情况下,对于交付的对象或用户创建的对象是不是包含缺陷就会模糊不清。如果对象的用户以某种特殊的方式重新定义了一套虚成员函数,由这段特殊代码产生的后续软件缺陷到底由谁来负责呢?有人认为,对象的生产者必须确保虚成员函数和继承不会在派生类中导致问题。也有人认为,多态使得预测每种对象误用是不可能的。随着软件组件的继续增殖,这种争论无疑还会持续。不过,提供注释准确的对象层次以及高质量的最终用户文档是必不可少的。如果对于对象接口的语义存在歧义,就应该通过完整的说明文档和高质量的源代码注释剔除这种歧义。

用于多线程环境的对象更是会碰到许多特殊的危险因素。在构建多线程对象时,期望线程规划策略和事件互斥量条件都有效是难以实现的。通过这本书,我们已经看到,通过的环境提供了与多线程处理相关的不同特征,但结果通常是相同的。得到这一结果的具体过程可能是产生难以捉摸软件缺陷的根源所在。如果对象的生产者以任务准则优先权为前提交付一个多线程对象,如果另一个不具有任务准则优先权的多线程对象继承了这一个对象,对象将如何操作呢?如果对象是线程安全的,但用于不安全的线程代码,可接受测试可能会产生不正确结果。在多线程环境中进行测试的固有难度可能会掩盖真正导致软件故障的组件。在这里,明智地使用 C++ 异常处理能力可以使将交付对象可靠,同时具有容错性。如果对象的生产者包括了必需的异常

处理，则执行可接受测试的用户将具备了找出导致软件故障的软件组件的基础，同时可以找出哪一个软件组件包含软件缺陷。多线程环境使得可接受测试特别复杂。强烈推荐在 C++ 中使用异常处理机制。

12.2 对象的测试实例

为了完成测试任务，我们定义测试实例作为代码执行序列，结合代码交互的数据以及代码执行的环境进行测试。没有解决代码数据和环境的测试实例不能提供代码执行的完整描述图。我们使用测试实例来测试和预测组件的性能。理想情况下，让测试实例代表目标环境中组件可能使用的每种方式。针对每种测试类别制定测试实例。这意味着必须针对单元测试、强度测试以及集成测试定义测试实例。每种测试类别都包含一套由测试实例和测试数据的测试内容。随着软件变得越来越复杂，同时它的用途也越来越广泛，制定耗尽测试实例是不切实际的。不过，在每种测试类别中，我们可以创建涵盖组件操作主要方面的实例。特别是在面向对象编程中，我们对评价以下方面的测试实例感兴趣：

- 对象构建；
- 对象析构；
- 对象赋值；
- 对象复制；
- 对象派生；
- 成员函数性能；
- 对象资源需求；
- 公有对象访问；
- 受保护对象访问；
- 线程化对象访问。

12.2.1 对象构建的测试实例

许多对象有多个构造函数。设计测试实例时，必须针对每个对象构造函数。此时，也必须测试用于防止继承的私有构造函数。对象的数据组件必须在每一个测试实例中进行检查。对象的构造函数是否正确初始化对象的所有组件？如果对象通过复合或继承包含其它对象，调用了正确的构造函数吗？所有这些对象都初始化了吗？在同一个进程中构建相同类型的多个对象时，必须制定测试实例来检查发生的事情。在多个进程中创建同一种类型的多个对象时，也要制定测试实例来检查发生的事情。例如有一个对象，它的构造函数打开一个多媒体设备。我们也许不能在一个进程中构建多个这样的对象。而且，在多个进程中构建多个这样的对象时，可能会发生问题。必须设计测试实例评价对象构建期间将发生什么——是否对象需要排它性访问某些设备，或共享访问某些设备。构建过程必须经受这样的测试。如果对象的构造函数打开或创建线程、互斥量、文件或设备，是否包含了异常处理？对于那些不仅仅赋值的对象构造函数必须包含异常处理器。构造函数失败时，异常处理是可以用于通信的最简单机制。

测试拷贝构造函数

还必须测试的一种特殊类型构造函数是拷贝构造函数 (copy constructor)，它拷贝一个对象。拷贝构造函数在复制对象的时候使用。这意味着在对象作为参数按值传递，或者某个参数是 return 语句中的返回值时，将调用拷贝构造函数。拷贝构造函数的参数是一个对它的类的对象的引用。与常规类型构造函数一样，通常不是显式调用拷贝构造函数。拷贝构造函数可以用于按值将对象的拷贝传递到某函数中，或者从某函数返回该对象。它也可以用于对象初始化。

当按值给函数传递对象时，编译器复制该对象并将它推入堆栈。拷贝构造函数指导编译器如何复制这个对象。如果没有定义拷贝构造函数，则编译器生成一个拷贝构造函数。生成的拷贝构造函数只是简单地逐个成员地复制数据值。这种复制方法称做浅层复制 (shallow copy)。如果对象为一个简单类，包含内置类型而不包含指针，这种复制方式是可以接受的。函数将使用该值与对象，而且它的行为不会受到更改。通过浅层复制，只复制成员的指针的地址，而不是复制地址指向的值。对象的数据值可能会被接收对象拷贝的函数无意间更改。当函数越出作用域时，对象的拷贝以及它所有的数据被弹出堆栈。如果对象有指针，需要执行深层复制 (deep copy)。通过对象的深层复制，在自由存储空间为对象分配内存，同时复制指向的元素。图 12-2 显示了浅层复制与深层复制之间的不同之处。

深层复制也能用于从一个函数返回对象。当在一个函数中声明对象时，将这个对象或数据成员的数据值复制给函数外部它的类的某个对象。复制构造函数用于将函数内的对象复制给函数外部的对象。必须制定测试对象复制构造函数的测试实例。这意味着这个测试要评价对象作用一个参数时的构建方式，以及用于 return 语句时的构建方式。复制构造函数的测试通常被忽视。因此，对象和对象层次的测试计划应该特别关注复制构造函数。

12.2.2 析构函数的测试实例

与对象的构造函数一样，必须制定专门的测试实例来检查对象析构期间的行为。许多对象在堆上动态分配存储空间。存储空间通常在对象的构造函数内分配。分配存储空间后，对象成员的数据值就保存在这个堆上。这种动态分配的存储空间一般则由对象的析构函数释放。一个常犯的错误是，在一个对象内声明多个指针，在对象的构造期间为每个指针分配动态存储空间，而在析构对象时，却不能成功释放这些存储空间。释放所有分配的内存失败是析构不成功的一般原因。例如，构造函数可能初始化 10 个指针，但析构函数只能释放 9 个指针的内存，使得析构对象后留下一个虚悬的指针。这是面向对象程序中发生内存泄漏的通常原因。因为对象越出了作用域，所以对象不再访问它所分配的内存。不过，如果它没有释放所有的内存，没有释放的内存就不能被进程中的其它对象使用。如果创建和销毁这样的对象（通常在进程的执行过程中），它就会吃掉内存，并最终导致系统崩溃。

如果析构函数用于关闭文件、释放互斥量、取消线程、销毁线程或关闭设备，则强烈推荐在析构函数内使用异常处理逻辑。因为析构函数没有返回值，对象的用户可能不知道析构函数操作已经失败。如果析构函数抛出一个异常，则要求用户在出现软件故障时采取一定类型的措施。如果对象的构造函数需要锁定某互斥量，对象的析构函数必须确保析构对象时该互斥量不再被锁定。不能取消锁定对象中锁定的每一个互斥量可能导致死锁或无限延迟。设计析构函数的测试实例时

必须要评价对象构造函数所需求的所有类型资源的析构和释放。

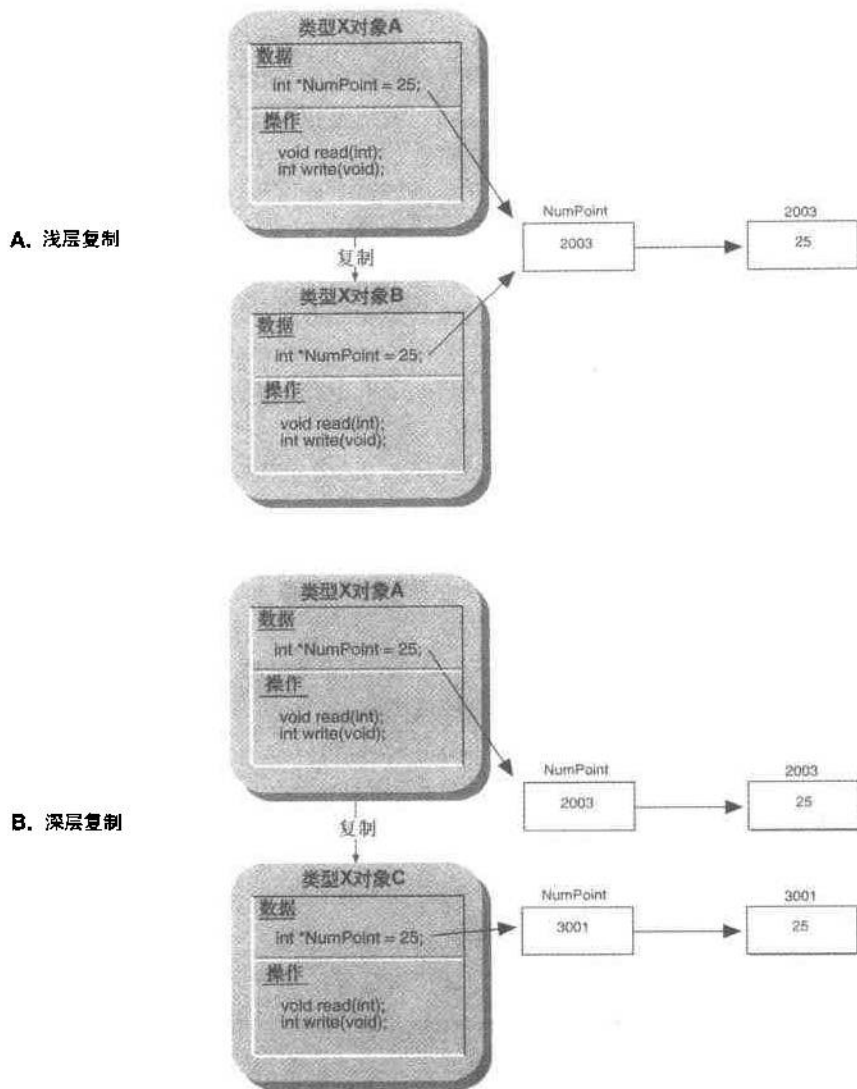


图 12-2 浅层复制与深层复制之间的区别

12.2.3 赋值的测试实例

并不是所有的对象都有赋值运算。这并不会阻止用户试图在赋值中使用对象。如果在赋值中使用对象没有意义，必须采取措施防止用户这样做。例如，为应用框架对象定义赋值运算符不会有用。在许多情况下，应用框架对象表示全体应用，所以在一个进程内具有多个拷贝的需要并不切合实际。在这种情况下，为了防止用户试图在赋值场合使用框架，赋值必须定义为空（null）、

受保护 (protected)、私有 (private) 等。如果对象不支持赋值, 当分配包含指针的对象时必须加以小心。图 12-2 描述了浅层复制与深层复制的行为。当赋值用于包含堆中对象指针的对象时, 情况与之相似。如果我们编写如下所示代码:

```
void main(void)
{
    Object A;
    Object B;
    B=A;
    create Thread1(B)
    ...
    ...
    ...
}
```

它创建了两个对象: A 和 B。如果对象 A 包含动态分配的数据, 将对象 A 赋予其它对象时必须加以小心。在这里是将对象 A 赋予对象 B。然后我们创建一个新线程并作为一个参数传递 B。这样做将导致一个问题的发生。甚至在 Thread1 执行前, 函数 main() 也可能越出作用域。对象 A 的析构函数可以释放为对象 A 创建的内存。如果对象 B 中的数据为此内存的一部分, 则 Thread1 会执行一个非法的内存访问。必须针对这种情形测试赋值运算符。而且, 如果对象占有互斥量、信号量或设备, 对于所涉及的对象, 赋值语句将呈现什么样的效果呢? 两个对象都占有资源吗? 两个对象都占有互斥量锁定吗? 显然, 这里并非如此。评价这种情况下赋值的对象性能的测试实例可能会相当有效。

对象复制

对象复制构造函数中存在的问题, 将一个对象复制到另一个对象的任何类型对象赋值中也同样存在相同的问题。如果源对象包含动态分配内存的指针, 同时通过复制构造函数、赋值运算符, 或者其它某种函数进行复制, 目标对象将包含什么呢? 目标对象是否实际创建了动态分配内存的拷贝, 或者拷贝只是简单包含动态内存指针的拷贝? 如果目标对象只包含指针, 则源对象和目标对象两者将指向相同的数据。如果目标对象创建数据的拷贝, 两个对象都将包含相同的源, 但这些值位于计算机中的不同地址。

12.2.4 对象派生子类

当派生类覆盖了虚成员函数或定义了纯虚成员函数, 或者基类通过引用操纵派生类的成员函数之时, 可能在语义上引入不可察觉的错误。当后代修改了任何虚函数的假定行为时, 评价在以上条件下发生的事的测试实例应当评价基对象或祖先对象将发生什么。C++ 异常处理机制此时也同样是处理在派生类中误导虚函数产生对象故障的最实际方式。当一个类使用虚成员函数和引用支持多态时, 这个类也必须包含保持类完整性的前置条件、后置条件以及断言逻辑。如果派生类中的函数试图违反类的断言、前置条件或后置条件, 则应该抛出异常。

12.2.5 成员函数性能的测试实例

显然, 类提供的每个成员函数的速度和效率也必须进行测试。强度测试最有可能测试成员函

数的性能。如果成员函数支持缺省值，而这些缺省值被类的用户覆盖，成员函数将如何操作？通过对对象成员函数的参数确保不将非法值传递给对象。如果成员函数由友元函数表示，必须制定测试实例评价非友元调用友元函数时的发生行为。也就是说，因为友元函数是一个自由漂浮的函数，它可能被随意调用。这些函数的性能在以上情况下必须是可预测的。而且，如果友元不仅仅是一种类型对象的友元，必须对它所关联的每一种对象评价友元函数的性能。

12.2.6 对象资源需求和测试实例

必须制定评价对象资源需求的测试实例。如果对象动态分配内存，对象可望合理地增大到多大？如果对象为多线程，对象需要的最少线程数是多少？对象要求对线程的最大数量有限制吗？如果目标环境不具备对象在它的作用域内所需求的所有资源，对象的性能如何？对象是否给系统返回一些未用的资源？对象迅速释放所需求的互斥量，还是在销毁对象前一直占据这些互斥量？对象需要如声卡、特殊视频卡或扫描仪等设备操作吗？如果对象不能获得这些设备，对象的行为将如何？必须设计针对以上情形的测试实例，而且在我们能够预测对象在相反情况下的行为之前，对象必须接受这些实例测试。

12.2.7 测试公有对象访问、受保护对象访问以及线程化对象访问

必须调用对象的用户可以访问的每个数据成员和成员函数。其理由似乎不言而喻。不过，通常那些很少使用的成员函数成为软件故障的原因。每个成员函数必须接受单元测试、强度测试以及集成测试。如果用户可以通过继承访问成员函数，或者因为成员函数是公有的，必须设计测试实例确保成员函数的正确操作。必须制定针对任意成员调用序列的测试实例。对象的所有公有或受保护成员函数应该能按任意顺序调用。因为没有任何因素可以阻止用户按任意顺序调用成员函数，所以必须用测试实例来判断任意调用成员函数时的情况。如果一个成员函数中获得了互斥量锁定，而在另一个成员函数中被取消锁定，当不按序列调用成员函数时将会发生什么呢？也就是说，如果首先调用取消锁定互斥量的成员时发生什么情况？必须测试不同类型的序列。而且，如果互斥量包含临界区，保护和公有访问策略支持什么样的并发模型？前面讲过，多线程环境中可以使用以下4种基本读/写访问模型：

- CREW——并发读排它性写。
- CRCW——并发读并发写。
- EREW——排它性读排它性写。
- ERCW——排它性读并发写。

在多线程应用程序中，如果成员函数用于访问对象的数据组件，公有和受保护成员函数应当适合于以上模型之一。而且，如果对象被多个线程并发、异步或同时使用，必须评价对象临界区的完整性。如果偶而有多线程访问对象，对象的临界区受保护吗？对一重进入代码，对象的行为如何？如果对象设计用于多线程环境，则必须设计涉及在单进程环境中被多个线程并发访问的对象的实例。而且在多处理器环境中被多个线程访问的对象也要设计相应的测试实例。为了预测对象在并发访问中的行为，必须创建使用多个线程和多个处理器的测试实例，而且对象必须接受不同数量线程和处理器的测试。

12.3 测试多线程架构的问题

测试多线程架构的问题是目标环境难以在测试场合模仿。在多线程环境中有两个主要原因：

- 继承和多态可以创建开放层次。
- 线程规划对上下文和系统负荷敏感。

12.3.1 开放层次问题

对象的生产者不能预测用户将来使用继承和多态来特殊化该对象的方式。如果生产者的对象有公有构造函数和虚成员函数，则用户可以在派生类中特殊化此对象。我们在前面从 `query_processor` 类创建了一个 `calculator` 类。`query_processor` 类的提供者可能没有预料到 `calculator` 类将作为它的一个后代。当派生类的身份未知时，测试基类是否可靠是一项复杂的工作。

12.3.2 规划问题

在多线程环境中，线程可能有不同的执行优先权。每种操作环境对待线程优先权都稍有不同。一些系统的线程具有高、中以及低优先权。一些系统有用户自定义的优先权级别。一些系统有任务准则优先权（mission-critical priority）、实时优先权（real-time priority）、普通优先权（normal priority）、背景优先权（background priority）等等。操作环境查看当前执行线程混合体、当前等待执行的线程混合体，然后基于优先权、负载平衡、资源可用性、处理器可用性、条件变量状态等等分派下一套线程。优先权向上或向下动态调整。具有相同优先权的两个线程可以按一种完全任意的方式分配给处理器。

例如，假如有一个名字列表，这些名字按字母顺序进行维护，刚好有 10 个人，它们都叫 John Smith，哪一个 John 排在第一位，哪一个 John 排在第二位，哪一个 John 排在第三位……？对于列表来说，这些名字间是没有区别的。然而，每个名字的确表示不同的人。如果这个列表在不同的场合进行不同的排序，列表的顶部可能出现的是不同的 John Smith。从排序的角度来看，这是没有差异的。同样，如果某进程连续执行了若干次，而且这个进程包含多个具有中等优先权的线程，每次执行进程时，可能按不同的顺序执行线程。对于分派器来说，同等对待所有的线程。不过，对于测试器来说，这就带来了问题，因为在这种情况下，我们不能复制测试环境。如果我们不能复制测试环境，我们就不能正确重现软件故障。如果我们不能重现软件故障，就不可能找出导致故障的原因。如果由于每个线程都具有相同的优先权，操作环境任意分派线程，此时就存在测试上的问题。

如果环境是一个动态环境，每次执行进程时，构建和析构不同的对象。如果架构以及架构存在的环境不断发生变化，而且由于继承和多态架构是末端开放的（open ended），那么，通过传统的方式证明程序的正确性极其困难。测试多线程应用程序的基础需要对说明性编程技术（declarative programming technique）、组合理论（combinatorial theory）、谓词演算（predicate calculus）以及逻辑语义（logic semantics）有充分的理解（这些内容已超出了本书的讨论范围）。不过，本书的重点是通过使用面向对象组件构建多线程架构来消除多种多线程编程上的缺陷。使用面向对

象组件来封装临界区、结合同步对象域对象，而且在可能的地方提供自动化锁定和取消锁定。通过使用面向对象软件构建方法，我们可以防止一定类型的错误发生。推荐正确选择软件模型和架构作为多线程程序中的第一道防线。而且，为软件选择正确的模型和架构后，测试计划和测试实例就能顺利制定，因为测试者具备了设计测试类别的基础。

12.4 使用常用模型和架构

测试模型用于系统验证。优秀建模和模拟技术的重要性在这里得到了体现。如果软件组件为真实思想的合理表达，则对真实世界特征和行为的理解可以应用于软件对应模拟体。这对于测试具有重要的含义。为了使测试计划成功，必须预先知道测试的结果。特别是，设计者必须指定要运行的测试、如何测试以及预料的结果是什么。如果软件组件是真实世界的良好模型或模拟，对于让设计者预先知道系统中类组件的行为就有帮助。例如，如果系统设计为一个银行建模，系统捕捉了银行的主要特征，不允许在午夜取款，除非在 ATM 机上进行！显然，这是一个简单化的例子。这里要说的是，设计者在创建测试模型时可以利用域知识。在域中相对于在软件中来说，参数、限制、系统常态以及不合理状态都更容易识别。如果在域中识别它们，它们可能在为域建模的类组件中更容易理解。

常用线程模型

对于多线程应用程序有多种常用模型：

- 主/次线程模型 (primary/secondary thread model)；
- 对等模型 (peer model)；
- 工作管道模型 (work pile model)。

每一种模型都将进程分成一套线程。当把应用程序分成一套进程时，进程间通常具有父-子关系，而包含多个线程的进程间没有这种关系。

一、主/次关系

通常在 main() 函数中创建的主线程将进程执行的工作分解成多个次线程。在主/次模型中，线程通常是分离的。这意味着它们只需要很少的通信或根本无需通信就可以一起同时执行。因为主/次模型将工作分解成基本上分离的任务，所以程序员不必担心复杂的同步情形。图 12-3 显示了具有主/次关系的一些线程间的关系。

主线程的功能只是创建次线程，一直等待到次线程完成为止，同时在需要时释放资源。主线程也可以用于管理死锁。如果主线程怀疑有死锁发生，主线程可能会取消违规的线程。除了创建和取消线程外，主线程与次线程之间的责任或契约很少。主/次模型大概是最易于使用的线程模型。不过要记住，不能将所有的进程都描述为一套交互很少或根本没有交互的分离线程。当主/次模型用于本书所描述的面向对象技术中时，就能顺利开发多线程应用程序。

二、同等模型

在同等模型中，由单个线程创建其它线程，但一旦创建了其它线程，创建者与进程中其它任何线程同等负责任务和操作。“同等模型 (peer model) 也称做工作同事模型 (work crew model)，

在这种模型中，在程序启动时，必须由一个线程创建其它所有对等线程……接下来这个线程只是充当另一个同等线程来处理请求，或者挂起自己等待另一个同等线程的完成”(Nichols 等, 1996)。同等模型倾向于比主/次模型需求更多的线程间通信。所以，同步化对同等模型的需要更多。当使用同等模型时，我们并没有假定进程可以分成一套分离线程。图 12-4 显示了同等模型中线程间的关系结构。

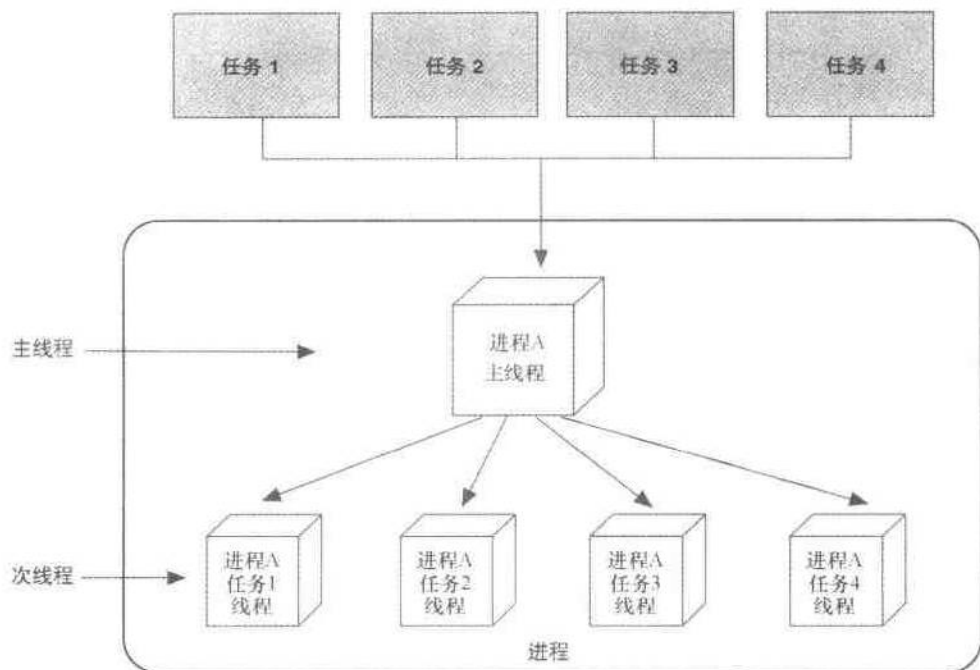


图 12-3 主和次线程间的关系。主线程的功能是创建次线程，一直等到它们完成，并在需要时释放资源

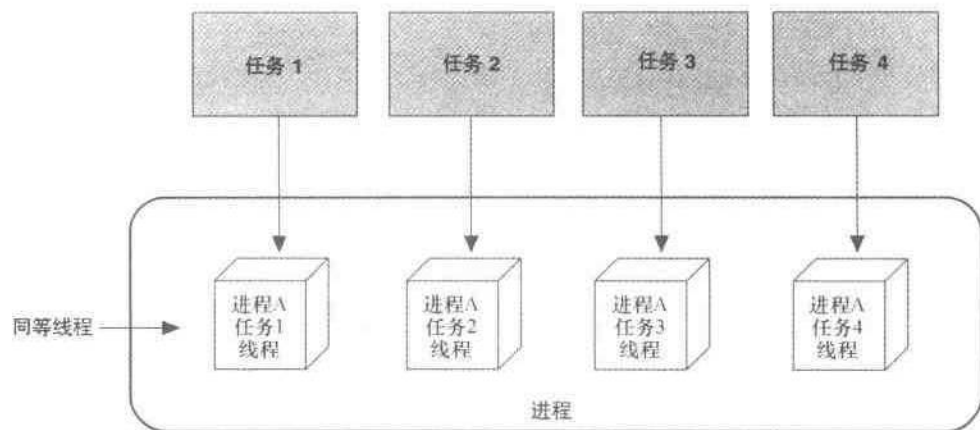


图 12-4 同等模型中线程间的关系结构

三、工作堆模型

工作堆模型是主/次模型的一种变种。“工作堆范例使用许多工作线程，每个线程都从中心堆请求一个工作分配。这个过程一直重复，直到没有更多的任务为止”（Kleiman, Shah, Smaalders, 1996）。工作堆模型也需要主/次线程间的通信、合作以及同步。图 12-5 显示了工作堆模型中的工作分工。

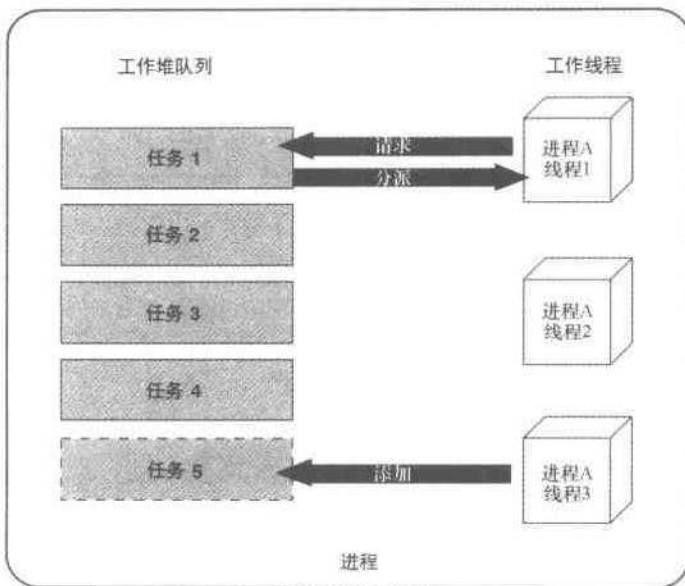


图 12-5 工作堆线程模型的结构。这个范例使用许多工作线程，每个线程从中心工作堆请求一个工作分配。工作堆是一个队列。这个过程一直重复，直到队列中没有更多的工作为止。

工作线程也可以在队列中添加工作

在同等模型中，没有假定工作管道模型中的线程是分离的。黑板结构可以通过线程工作管道模型的变种来实现。使用线程工作堆模型可以实现像动态编程之类的高级编程技术。使用常见线程模型实现多线程应用程序，程序员可以利用每种模型具备的已知优势与弱点。

实现并发的最后思考

最后，在调查研究我们所引导的自然加速运动过程中，就像用手操作一样，遵循自然界本身的习惯与规律，在她的所有其它过程中，仅应用那些最常见、最简单和最容易的方式。

Mathematics of Motion

—— Galileo Galilei

可以分解成并发执行进程的应用程序以及可以分解成并发执行线程的进程为软件开发带来了设计、实现和测试上的特殊问题。准备在应用程序中使用并发和并行的程序员和开发者可能遇到 5 种常见的缺陷：

- 竞争条件（数据竞争）；
- 无限延迟；
- 资源死锁（deadlock 或 deadly embrace）；
- 规划交错锁定（gridlock）（即优先权倒置）；
- 脆弱架构。

有许多非面向对象方式来识别和避免支持并发应用程序中的缺陷。一些经典算法，如 Dijkstra 的 Banker 算法、Lamport 的 bakery 算法、Dekker 算法以及 Petterson 算法，早就尝试使用信号量、互斥量、程序监视量以及死锁避免来解决并发问题。这些算法的确有助于说明这些问题及其解决方案。对于 Solaris、Windows NT 以及 OS/2 等现代商业操作系统，现在程序员都可以用来在应用程序中实现并发的 API。这些 API 包括能够在线程间广播通信的时间耗尽参数以及条件变量的互斥量。现代操作系统提供的线程处理以及互斥量 API 的特征使得程序员和软件开发者在迎接并发编程挑战的道路上又向前迈进了一步。

在本书中，我们为异步、并行以及并发编程添加了一道新的防线。我们的重点是架构。我们使用 C++ 组件和面向对象技术来构建多线程架构。使用封装为程序中的临界区提供了保护。我们使用封装为提供线程处理和同步化的操作系统 API 提供面向对象接口。使用封装和接口类，探讨

了互斥量对象、事件对象、线程对象、条件变量对象、命名管道流以及匿名管道流的使用。本书提倡应用程序中的临界区应当封装在一个类中。每个类应该通过继承或复合包含保护临界区必需的必要同步对象。而且，互斥量的锁定和取消锁定应该由类的成员函数自动完成。本书还向读者演示自由漂浮信号量、互斥量以及条件变量都是导致灾难的根源。通过面向对象技术的合理应用演示了在多线程应用程序中解决问题的一种切实可行的途径。

当面向对象技术发挥作用时，对数据的所有访问都通过对象成员函数（也称做方法）来调控。对象成员函数充当用户与对象所包含数据之间的接口。建议将同步化线程和保护临界区的责任由用户身上转移到对象的提供者身上。建议按对象成员函数决定外界如何、何时以及访问哪一部分数据的相同方式，让对象的成员函数也决定在多线程环境中访问数据的方式。一旦可以在多线程环境中访问对象的数据，就有可能产生竞争条件。这是为什么将同步化工作从对象外部移到对象内部的原因。如果我们让对象的用户承担对象同步化的责任，就违背了面向对象设计的基本原则。因此，作为面向对象策略，我们将同步变量、条件变量、事件信号量以及线程创建移到了对象的类中。这样可以完全控制与对象有关的并发问题。

我们使用接口类为非面向对象函数和数据提供面向对象接口，这一技术贯穿本书的始末。本书使用接口类为 POSIX 线程 API、Win32 线程 API 以及 OS/2 线程 API 提供面向对象接口实例类。建议程序员和软件开发者不要按 API 的原始、自由漂浮形式使用它们。通过为 STL 容器和算法提供接口类，我们决定牺牲灵活性换取安全性。虽然 STL 容器和算法提供了强大的编程解决方案，但它们违背了一些重要的面向对象编程基础，而且它们有可能在多线程环境中适得其反。通过提供适配 STL 接口的接口类，我们可以改进 STL 容器和算法，使它们在多线程环境中更安全。我们的目标是处理非线程安全的每个数据组件及函数或过程，并将组件封装在一个类中。一旦将组件封装在类中，我们就可以用面向对象互斥量和面向对象条件变量围绕对组件的所有访问。本书重点讨论了增量多线程处理方式，它是使用 C++ 组件作为基石来驱动的。我们拥有支持通信的基石，如面向对象命名管道；还拥有支持同步的基石，如面向对象互斥量。结合这些基石与其它 C++ 组件，如 iostream 对象、STL 容器以及 STL 算法。我们探讨了结合同步对象与宿主类的概念。每个宿主类拥有了适当的同步对象后，我们就有了线程安全基石。

根据作者的经验，使用线程安全基石的增量多线程处理避免和消除了多个线程或多个进程并发执行情况下碰到的许多缺陷。脆弱架构、数据竞争以及死锁等问题都可以通过面向对象编程技术有效得以解决。线程间存在 4 种基本同步关系：

- SS——Star-to-Start
- SF——Start-to-Finish
- FS——Finish-to-Start
- FF——Finish-to-Finish

以上关系主宰着共享对象和数据线程间的交互作用。理解这些关系如何影响多线程应用程序是成功实现、维护、改进和测试应用程序的关键。以上关系可以用于多线程应用程序的说明文档中。而且，我们使用线程依赖性矩阵和线程依赖性图作为说明多个并发执行线程间的关系。

我们在解决竞争条件、无限延迟以及死锁问题的过程中应用了两个基本思想。第一个思想是使用面向对象基石的增量多线程处理。第二个基本思想是选择能自然适应于多个线程的架构。作

者主张设计和实现多线程应用程序的有效途径应该是以选择能自然适合于并行化的架构为起点。这意味着，这些架构应该具有隐式或显式的内在并行性。满足这些标准的常用架构有 3 种：

- 客户机/服务器；
- 事件驱动；
- 黑板。

以上每一种架构及其变种在它们的组件间都具有内在的并行性。我们发现，挑选出能自然适应并发的架构，与试图在实际中最好实现为单个线程应用程序的应用架构中强加多个线程相比，前者更理想。我们的目标是结合支持并发架构的选择与增量多线程处理技术及面向对象编程技术来得到多线程面向对象架构的思想。我们提倡的是使用面向对象架构实现并发，而不是通过算法实现并发。通过选择实现并发的架构，可以决定让功能服从形式。首先挑选出架构，然后选出架构支持的逻辑。我们推荐实现并发的途径应该摒弃以过程为中心的技术（*procedure-centered technique*），而应该选择以数据为中心的技术（*data-centered technique*），因为让我们陷入麻烦的正是临界区。虽然本书试图讨论尽量多的内容，但只是涉及了多线程和并行编程的一些问题表面。我们为读者提供了参考书目，其中列出了许多可以为读者提供更全面、更高级多线程和并行处理技术的资源。我们希望本书为 C++ 程序员至少提供一个构建多线程环境中对象的方向。

附录 A POSIX 线程管理规范

第 13 节 执行规划

§ 13.1 规划参数

规划参数结构 `sched_param` 包含实现每种支持规划策略所需要的规划参数 (scheduling parameter)。它在 `<sched.h>` 中定义, 至少包含表 1 所示成员。

表 1 定义规划参数所包含成员

成员类型	成员名字	描 述
int	<code>sched_priority</code>	进程执行规划优先权

允许按 § 1.3.1.1, 第(2)条的要求在实现中添加扩展。为这个结构添加扩展后, 与该结构中字段何时为未初始化有关的应用程序行为可能会发生改变, 因此也要求能对这部分进行相应的扩展, 参见 § 1.3.1.1 的要求。

包含 `<sched.h>` 头后, 让本标准允许使用的符号在头 `<time.h>` 中可见。

§ 13.2 规划策略

在本节中所描述的规划语义是通过包含一套线程列表的概念性模型来定义的。使用这种概念性模型并不必需实现结构。它假定在本模型描述的操作过程中没有时间流逝, 所以不可能有同时操作。这种模型只讨论对于可运行线程的处理器规划, 但必须注意, 如果其它资源序列考虑了处理器规划, 将使实时应用程序的可预测性大大提高。

从概念上讲, 对于每个优先权存在一个线程列表。任何可运行线程都可能位于任何线程列表中。要提供多个规划策略。排序每个非空线程列表, 包含一个头作为其排序的末端之一, 一个尾

部作为另一个末端。规划策略的目的是定义这套列表上的允许操作（例如，在列表间移动线程）。

每个进程由关联的规划策略和优先权控制。这些参数可以通过 `sched_setscheduler()` 或 `sched_setparam()` 函数的显示应用执行来定义。

每个线程由关联的规划策略和优先权控制。这些参数可以通过 `pthread_setschedparam()` 函数的显示应用执行来定义。

与每个策略关联的是一个优先权范围。每个策略定义要指定该策略的最小优先权范围。每个策略的优先权范围可以与其它策略和优先权范围重叠，也可以不重叠。

相应的实现应该选择如下定义的线程：位于即将成为运行线程的最高优先权非空线程列表的头处，而不管它的关联策略如何。然后将这个线程从线程列表中删除。

明确需要 3 种规划策略，其它规划策略由实现定义。表 2 中的符号在头 `<sched.h>` 中定义。

表 2 在头 `<sched.h>` 中定义的符号

符 号	描 述
SCHED_FIFO	先进先出（FIFO）规划策略
SCHED_RR	轮询规划策略
SCHED_OTHER	另一个规划策略

这些符号的值互不相同。

§ 13.2.1 SCHED_FIFO

相应实现包含一个称做 FIFO 规划策略的规划策略。

在此策略下规划的线程是从一个其中线程没有执行的排序线程列表中挑选出来的；一般而言，列表的头为在列表中存在时间最久的线程，尾部为在列表中存在时间最短的线程。

依据 SCHED_FIFO 策略，定义线程列表的修改如下：

（1）当某运行线程变成抢占线程时，它成为该优先权线程列表的头。

（2）当阻塞线程变成可运行线程时，它成为该优先权线程列表的尾部。

（3）当运行线程调用 `sched_setscheduler()` 函数时，在函数调用中指定的进程针对指定策略和 `param` 参数指定的优先权进行修改。

（4）当运行线程调用 `sched_setparam()` 函数时，在函数调用中指定的优先权修改为 `param` 参数指定的优先权。如果修改了优先权的线程为一个运行线程或为可运行线程，可运行线程成为新优先权线程列表中的尾部。

（5）当运行线程调用 `pthread_setschedparam()` 函数时，在函数调用中指定的线程修改为由 `param` 参数指定的策略和优先权。

（6）如果一个策略和优先已经被修改过的线程是一个运行线程或可运行线程，则可运行线程成为新优先权线程列表的尾部。

（7）当运行线程调用 `sched_yield()` 函数时，该线程变成对应优先权线程列表的尾部。

(8) 其它场合, 线程列表内线程的位置和它的规划策略不会受到影响。

对于这种策略, 有效优先权应介于函数 `sched_get_priority_max()` 和 `sched_get_priority_min()` 将 `SCHED_FIFO` 作为参数返回的范围之中。相应的实现要针对这种策略提供至少 32 种优先权的优先权范围。

§ 13.2.2 SCHED_RR

相应的实现包含一个称做轮询规划策略 (round-robin scheduling policy) 的规划策略。这种策略与 `SCHED_FIFO` 策略在以下附加条件是相同的: 当实现发现一个运行线程已经作为运行线程执行的时间等于或超过函数 `sched_rr_get_interval()` 返回的时间段时, 该线程将成为其线程列表的尾部, 而且删除线程列表的头, 构成一个运行线程。

这种策略的作用是确保同一优先权存在多个 `SCHED_RR` 线程时, 其中的线程不会独占处理器。如果应用程序在相同或较高优先权级别使用 `SCHED_FIFO` 策略, 或者在较高优先权级别包含多个线程, 就不应该只依赖于使用 `SCHED_RR` 来确保在多个线程中应用程序的进度。

使用这种策略被抢占, 然后作为运行线程恢复执行的线程将完成未过期轮询时间段。

对于这种策略, 有效优先权应位于函数 `sched_get_priority_max()` 和 `sched_get_priority_min()` 以 `SCHED_RR` 为参数的返回范围之中。相应的实现将提供一个针对该策略至少 32 个优先权的优先权范围。

§ 13.2.3 SCHED_OTHER

相应的实现将包含一个标识为 `SCHED_OTHER` 的规划策略 (它可以与 `FIFO` 或轮询规划策略同等执行)。相应实现要为规划策略定义中所描述的策略行为作出诠释。系统中的其它线程通过 `SCHED_FIFO` 或 `SCHED_RR` 执行, 利用 `SCHED_OTHER` 策略的规划线程作用要通过实现定义。定义该策略, 让严格一致的应用程序能够表明它们不再需要可移植方式的实时规划策略。

对于此策略下的执行线程, 实现只使用函数 `sched_get_priority_max()` 和 `sched_get_priority_min()` 以 `SCHED_OTHER` 为参数的返回范围之中的优先权。

§ 13.3 进程规划函数

§ 13.3.1 设置规划参数

函数: `sched_setparam()`

§ 13.3.1.1 纲要

```
#include < sched.h>
```

```
int sched_setparam()(pid_t pid, const struct sched_param *param);
```

§ 13.3.1.2 描述

如果定义了 `{_POSIX_PRIORITY_SCHEDULING}`:

`sched_setparam()` 将 `pid` 指定的进程规划参数设置为由 `param` 指向的 `sched_param` 结构指定的源。 `param` 结构中 `sched_priority` 成员的值为 `pid` 所指定的当前进程规划策略所包含范围内的任何一个整数。对应于优先权的较高数值表示较高的优先权。如果 `pid` 的值为负值, 则 `sched_setparam()`

函数的行为为未定义。

如果存在由 `pid` 指定的进程，而且调用进程拥有权限，则规划参数针对进程 ID 等于 `pid` 的进程而设置。

如果 `pid` 为 0，规则参数将针对调用进程进行设置。

一个进程拥有修改另一个进程规划参数的权限的条件通过实现定义。

实现可以要求请求进程具有适当的权限来设置它自己的或另外进程的规划参数。

不管目标参数是否正在运行，它都在其它所有相同或更高优先权可运行进程规划进行运行后恢复。

如果由 `pid` 参数指定的进程优先权设置为高于最低优先权的运行进程，而且指定进程准备运行，则由 `pid` 参数指定的进程将抢占一个最低优先权运行进程。同样，如果调用 `sched_setparam()` 的进程设置它的优先权低于其它一个或多个非空进程列表的优先权，则位于最高优先权列表中头的进程也抢占该调用进程。因此，对于以上任一种情况，原始进程都不能接收到请求优先权更改完成的通知，直到较高优先权进程执行后为止。

如果 `pid` 指定进程的当前规划策略不是 `SCHED_FIFO` 或 `SCHED_RR`，还包括 `SCHED_OTHER`，则其结果由实现定义。

这个函数在单个线程上的效果取决于该线程的规划竞争作用域。对于具有系统规划竞争作用域的线程，这些函数对于它们的规划无影响。对于具有进程竞争范围的线程，与其它进程中线程相关的规划可能依赖于特殊进程的规划策略和规划参数，该特殊进程使用这些函数来控制。

否则：

或者由实现支持 `sched_setparam()` 函数（如上面所描述的），或者 `sched_setparam()` 函数失败。

§ 13.3.1.3 返回值

如果成功，`sched_setparam()` 函数返回 0。

如果 `sched_setparam()` 调用未成功，优先权保持不变，同时函数返回值 -1，并设置 `errno`，让它指示该错误。

§ 13.3.1.4 错误

若发生下面任何一种条件，`sched_setparam()` 函数将返回 -1，并将 `errno` 设置成相应值：

[EINVAL] 请求规划参数的其中一个或多个位于指定 `pid` 规划策略定义的范围之外。

[ENOSYS] 实现不支持函数 `sched_setparam()`。

[EPERM] 请求进程没有针对指定进程设置规划参数的权限，或者没有调用 `sched_setparam()` 的相应权限。

[ESRCH] 没有对应于 `pid` 指定的进程。

§ 13.3.1.5 交叉参考

`sched_getparam()`，§ 13.3.2；`sched_getscheduler()`，§ 13.3.4；`sched_setscheduler()`，§ 13.3.3。

§ 13.3.2 获取规划参数

函数：`sched_getparam()`

§ 13.3.2.1 纲要

```
#include <sched.h>
int sched_getparam(pid_t pid, struct sched_param *param);
```

§ 13.3.2.2 描述

如果定义了{`_POSIX_PRIORITY_SCHEDULING`}:

`sched_getparam()`函数将返回在 `param` 指向的 `sched_param` 结构中 `pid` 指定进程的规划参数。

如果 `pid` 指定进程存在, 而且调用进程拥有相应权限, 则返回进程 ID 等于 `pid` 的进程的规划参数。

如果 `pid` 等于 0, 则返回调用进程的规划参数。如果 `pid` 的值为负, 则 `sched_getparam()`函数的行为未定义。

否则:

或者实现支持 `sched_getparam()`函数 (如上所描述), 或者 `sched_getparam()`函数失败。

§ 13.3.2.3 返回值

`sched_getparam()`成功完成时, 它返回 0。如果 `sched_getparam()`调用未成功, 则函数返回-1, 同时设置 `errno` 表征这个错误。

§ 13.3.2.4 错误

如果发生下面条件之一, `sched_getparam()`函数将返回-1, 并将 `errno` 设置成相应值:

[`ENOSYS`] 实现不支持函数 `sched_setparam()`。

[`EPERM`] 请求进程没有获取指定进程规划参数的权限。

[`ESRCH`] 没有对应于 `pid` 指定的进程。

§ 13.3.2.5 交叉参考

`sched_getscheduler()`, § 13.3.4; `sched_setparam()`, § 13.3.1; `sched_setscheduler()`, § 13.3.3。

§ 13.3.3 设置规划策略和规划参数

函数: `sched_setscheduler()`

§ 13.3.3.1 纲要

```
#include <sched.h>
int sched_setscheduler(pid_t pid, int policy,
                      const struct sched_param *param);
```

§ 13.3.3.2 描述

如果定义了{`_POSIX_PRIORITY_SCHEDULING`}:

`sched_setscheduler()`函数分别将 `pid` 指定进程的规划策略设置成 `policy`, 将 `param` 指向的 `sched_param` 结构中指定的参数设置成 `param`。 `param` 结构中 `sched_priority` 成员的值 of `policy` 指定的规划策略包含优先权范围内的任何整数。如果 `pid` 的值为负数, 则 `sched_setscheduler()`函数的行为未定义。

`policy` 参数的可能值在头文件<`sched.h`>中定义。

如果 `pid` 指定进程存在, 而且调用进程拥有对应权限, 则针对进程 ID 等于 `pid` 的进程设置规划策略和规划参数。

如果 `pid` 为 0, 则为调用进程设置规划策略和规划参数。

某个进程具有更改另一个进程规划参数的对应权限的条件在实现中定义。

实现可能要求请求进程具有设置它自身规划参数或另一个进程规划参数的权限。另外，对于设置进程自身规划策略，或另一个进程规划策略为特殊值所需要的相应权限可以应用实现定义的限制来达到目的。

如果 `sched_setscheduler()` 函数成功分别将 `pid` 指定进程的规划策略和规划参数设置成 `policy` 和结构 `param` 指定的值，则认为 `sched_setscheduler()` 函数成功执行。

这个函数对每个线程的作用取决于线程的规划竞争作用域。对于具有系统规划竞争作用域的线程来说，这些函数对它们的规划没有影响。对于具有进程规划竞争作用域的线程来说，在其它进程中与线程有关的规划可能依赖于它们的特殊进程的规划策略和规划参数，这个特殊进程使用这些函数来控制。

否则：

或者实现支持 `sched_setscheduler()` 函数（如上所描述），或者 `sched_setscheduler()` 函数失败。

§ 13.3.3.3 返回值

函数成功完成之时，它将返回指定进程的前一个规划策略。如果 `sched_setscheduler()` 不能成功完成，策略和规划参数将保持不变，同时函数返回值 -1，并设置 `errno` 表征该错误。

§ 13.3.3.4 错误

如果发生以下条件之一，`sched_setscheduler()` 函数将返回 -1，并将 `errno` 设置成对应值：

[EINVAL] `policy` 参数的值无效，或者包含在 `param` 中的一个或多个参数越出了指定规划策略的有效范围。

[ENOSYS] 实现不支持函数 `sched_setscheduler()`。

[EPERM] 请求进程没有设置指定进程规划参数或（和）规划策略的权限。

[ESRCH] 没有对应于 `pid` 指定的进程。

§ 13.3.3.5 交叉参考

`sched_getparam()`, § 13.3.2; `sched_getscheduler()`, § 13.3.4; `sched_setparam()`, § 13.3.1。

§ 13.3.4 获取规划策略

函数：`sched_getscheduler()`

§ 13.3.4.1 纲要

```
#include <sched.h>
```

```
int sched_getscheduler(pid_t pid);
```

§ 13.3.4.2 描述

如果定义了 `{_POSIX_PRIORITY_SCHEDULING}`：

`sched_getscheduler()` 函数将返回 `pid` 指定进程的规划策略。如果 `pid` 为负值，则 `sched_getscheduler()` 函数的行为未定义。

`sched_getscheduler()` 可以返回的值在头文件 `<sched.h>` 中定义（参见 `sched_setscheduler()`）。

如果 `pid` 指定进程存在，而且调用进程拥有相应权限，则返回进程 ID 等于 `pid` 的进程的规划策略。

如果 `pid` 为 0, 则返回调用进程的规划策略。

否则:

或者实现支持 `sched_getscheduler()` 函数 (如上所描述), 或者 `sched_getscheduler()` 函数失败。

§ 13.3.4.3 返回值

`sched_getscheduler()` 函数成功完成, 它返回指定进程的规划策略。如果未成功, 函数将返回 -1, 将设置 `errno` 表征该错误。

§ 13.3.4.4 错误

如果发生以下条件之一, `sched_getscheduler()` 函数将返回 -1, 并将 `errno` 设置成相应值:

[ENOSYS] 实现不支持函数 `sched_getscheduler()`。

[EPERM] 请求进程没有设置决定指定进程规划策略的权限。

[ESRCH] 没有对应于 `pid` 指定的进程。

§ 13.3.4.5 交叉参考

`sched_getparam()`, § 13.3.2; `sched_setparam()`, § 13.3.1; `sched_setscheduler()`, § 13.3.3.

§ 13.3.5 放弃处理器

函数: `sched_yield()`

§ 13.3.5.1 纲要

```
#include <sched.h>
int sched_yield(void);
```

§ 13.3.5.2 描述

如果至少定义了 `{_POSIX_PRIORITY_SCHEDULING}` 或 `{_POSIX_THREADS}` 的其中一个:

`sched_yield()` 函数强迫运行线程放弃处理器, 直到它再次成为线程列表的头。

否则:

或者实现支持 `sched_yield()` 函数 (如上所描述), 或者 `sched_yield()` 函数失败。

§ 13.3.5.3 返回值

如果 `sched_yield()` 函数成功完成, 它返回 0, 否则返回值 -1, 同时设置 `errno` 表征该错误。

§ 13.3.5.4 错误

如果发生以下条件之一, `sched_yield()` 函数将返回 -1, 并将 `errno` 设置成相应值:

[ENOSYS] 它的实现不支持函数 `sched_yield()`。

§ 13.3.6 获取规划参数极限值

函数: `sched_get_priority_max()`、`sched_get_priority_min()`、`sched_rr_get_interval()`

§ 13.3.6.1 纲要

```
#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_rr_get_interval(pid_t pid, struct timespec *interval);
```

§ 13.3.6.2 描述

如果定义了 `{_POSIX_PRIORITY_SCHEDULING}`:

`sched_get_priority_max()` 和 `sched_get_priority_min()` 函数分别返回对应于 `policy` 指定规划策略

的相应最大值或最小值。sched_rr_get_interval()函数更新 interval 参数引用的 timespec 结构，让它包含 pid 指定进程的当前执行时间极限（即时间片）。如果 pid 为 0，则返回调用进程的当前执行时间极限。

policy 的值为<sched.h>中定义的规划策略值之一。

否则：

或者实现支持 sched_get_priority_max()、sched_get_priority_min()和 sched_rr_get_interval()函数（如上所描述），或者 sched_get_priority_max()、sched_get_priority_min()和 sched_rr_get_interval()每个函数都失败。

§ 13.3.6.3 返回值

如果成功，sched_get_priority_max()和 sched_get_priority_min()函数将分别返回相应的最大值与最小值。如果未成功，它们将返回-1 并设置 errno 表征该错误。

§ 13.3.4.4 错误

如果发生以下条件之一， sched_get_priority_max()、 sched_get_priority_min() 和 sched_rr_get_interval()函数将返回-1，并将 errno 设置成相应值：

[EINVAL] policy 参数值不表示一个定义的规划策略。

[ENOSYS] 实现不支持函数 sched_get_priority_max()、 sched_get_priority_min() 和 sched_rr_get_interval()。

[ESRCH] 没有对应于 pid 指定的进程。

§ 13.3.6.5 交叉参考

sched_getparam(), § 13.3.2; sched_setparam(), § 13.3.1; sched_getscheduler(), § 13.3.4; sched_setscheduler(), § 13.3.3。

§ 13.4 线程规划

本条定义了一套为单个 POSIX.1 进程和 POSIX.1 系统内多个线程的规划编程控制提供广泛接口的操作。在本节中，规划接口与规划策略分离，同时由支持线程的实现提供某些函数。不过，不要求实现支持任何特殊规划策略。特殊规划策略由其它选项来指定，这些选项包括一套基于优先权的策略。规划策略的支持，包括由此标准定义的可选基于优先权的策略，包含对第 11 节所描述的规划相关属性和同步机制函数的支持。本节也描述了这些规划相关同步属性和函数。因此，没有对所描述特殊规划策略或依赖于规划策略的同步属性与函数的支持，实现也可以支持规划接口。

§ 13.4.1 线程规划属性

在对规划线程的支持中，线程具有通过 pthread_attr_t 线程创建属性对象访问的一些属性。

contentionscope 属性将线程的规划竞争作用域定义为 PTHREAD_SCOPE_PROCESS 或 PTHREAD_SCOPE_SYSTEM。

inheritsched 属性指定某新创建的线程是否继承创建线程的规划属性，或者根据 pthread_attr_t 对象中的其它规划属性拥有自己的规划值设置。

schedpolicy 属性定义线程的规划策略。schedparam 属性定义线程的规划参数。某进程中具有

不同策略的线程的交互将作为这些策略的一部分进行描述。

如果定义了 `{_POSIX_THREAD_PRIORITY_SCHEDULING}` 选项, 而且 `schedpolicy` 属性指定了此选项下的其中一个基于优先权的策略, `schedparam` 属性包含该线程的规划优先权。当使用线程属性对象来创建线程, 或动态修改线程的规划属性时, 相应的实现确保 `schedparam` 中的优先权值位于与规划策略相关的范围之内。 `schedparam` 中优先权值的意义与 § 13.2 中定义的 `priority` 的意义相同。

当创建一个进程时, 它的单个线程具有的规划策略与相关属性等于进程的策略和属性。缺省规划竞争作用域值由实现定义。其它规划属性缺省值由实现定义。

§ 13.4.2 规划竞争作用域

线程的规划竞争作用域定义了一套线程, 该线程必须与这套线程竞争使用处理资源。在某时间某执行点, 在每个处理器上, 规划操作最多选择一个线程来执行, 而且无论是否在进程规划竞争作用域下或在系统规划竞争作用域下, 该线程的规划属性 (如优先权) 是用于决定规划决策的参数。

在规划一个混合范围环境的上下文中, 规划竞争作用域对线程的影响如下:

——通过 `PTHREAD_SCOPE_SYSTEM` 规划竞争作用域创建的线程与同一规划分配域 (与系统规划属性相关) 中的其它所有线程一同竞争资源。通过 `PTHREAD_SCOPE_SYSTEM` 规划竞争作用域创建的线程系统规划属性是创建该线程所使用的属性。通过 `PTHREAD_SCOPE_PROCESS` 规划竞争作用域创建的线程系统规划属性是由实现定义的到规划属性的系统属性空间的映射, 这些规划属性在创建该线程时使用。

——通过 `PTHREAD_SCOPE_PROCESS` 规划竞争作用域创建的线程直接与通过 `PTHREAD_SCOPE_PROCESS` 规划竞争作用域创建的进程内的其它线程进行竞争。这种竞争基于线程的规划属性和策略来解决。未定义如何相对于其它进程或具有 `PTHREAD_SCOPE_SYSTEM` 规划竞争作用域的线程进行规划这种线程。

——相应实现将支持 `PTHREAD_SCOPE_PROCESS` 规划竞争作用域, 或 `PTHREAD_SCOPE_SYSTEM` 规划竞争作用域, 或者两者都支持。

§ 13.4.3 规划分配域

实现将支持包含一个或多个处理器的规划分配域。应当注意, 多处理器的存在并不自动表示规划分配域大小大于 1。在多处理器上的实现可能映射 CPU 的所有或部分子集到一个或多个规划分配域。它们可以基于每个线程、每个进程或每个系统定义这些规划分配域, 这取决于实现所支持的应用程序的类型。规划分配域与规划竞争作用域无关, 规划竞争作用域只定义一套该线程必须与一同竞争处理器资源的线程一样, 但规划分配域定义一套它所竞争的处理器。针对多个处理器在多个线程间解决竞争的语义由这些线程的规划策略来决定。

规划分配域大小以及对于规划分配域的应用控制级由实现来定义。相应实现可以在任何时候改变规划分配域的大小以及把线程绑定到规划分配域上。

对于规划分配域大小等于 1 的应用程序线程, 应当使用 § 13.2 中为 `SCHED_FIFO` 和

SCHED_RR 定义的规划规则。具有系统规划竞争作用域的所有线程，不管它位于哪一个进程中，它们都根据优先权来竞争处理器。具有进程规划竞争作用域的进程仅与它们的进程内其它具有进程规划竞争作用域的线程竞争。

对于规划分配域大小大于 1 的应用程序线程，以实现定义的方式使用 § 13.2 中为 SCHED_FIFO 和 SCHED_RR 定义的规则。每个具有系统规划竞争作用域的线程根据其优先权，按实现定义的方式在它的规划分配域中竞争处理器。规划具有进程规划竞争作用域的线程与该进程相同规划竞争作用域中的其它线程相关。

§ 13.4.4 规划说明

如果定义了 {_POSIX_THREAD_PRIORITY_SCHEDULING}，则 SCHED_OTHER、SCHED_FIFO 和 SCHED_RR 之外的规划策略以及由其它值表征的规划策略作用、支持这一策略所需的属性都通过实现定义。而且，实现将诠释支持这些策略的所有处理器规划分配域值。

§ 13.5 线程规划函数

§ 13.5.1 线程创建规划属性

函数：pthread_attr_setscope(), pthread_attr_getscope(), pthread_attr_setinheritsched(), pthread_attr_getinheritsched(), pthread_attr_setschedpolicy(), pthread_attr_getschedpolicy(), pthread_attr_setschedparam(), pthread_attr_getschedparam()

§ 13.5.1.1 纲要

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr,
    int contentionscope);
int pthread_attr_getscope(const pthread_attr_t *attr,
    int *contentionscope);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
    int *inheritsched);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
    int *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr,
    const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
    struct sched_param *param);
```

§ 13.5.1.2 描述

如果定义了 {_POSIX_THREAD_PRIORITY_SCHEDULING}:

pthread_attr_setscope() 和 pthread_attr_getscope() 函数用于设置和获取 attr 对象中的 contentionscope 属性。contentionscope 属性可能具有表示系统规划竞争作用域的值

`PTHREAD_SCOPE_SYSTEM`，或者具有表示进程规划竞争作用域的值 `PTHREAD_SCOPE_PROCESS`。符号 `PTHREAD_SCOPE_SYSTEM` 和 `PTHREAD_SCOPE_PROCESS` 由头 `<pthread.h>` 定义。

函数 `pthread_attr_setinheritsched()` 和 `pthread_attr_getinheritsched()` 设置和获取 `attr` 参数中的 `inheritsched` 属性。

当线程属性对象被 `pthread_create()` 使用时，`inheritsched` 属性决定如何设置创建线程的其它规划属性：

PTHREAD_INHERIT_SCHED 指定规划策略和相关属性从创建线程中继承，同时忽略 `attr` 参数中的规划属性。

PTHREAD_EXPLICIT_SCHED 指定规划策略和相关属性设置为该属性对象的对应值。

符号 `PTHREAD_INHERIT_SCHED` 和 `PTHREAD_EXPLICIT_SCHED` 在头 `<pthread.h>` 中定义。

函数 `pthread_attr_setschedpolicy()` 和 `pthread_attr_getschedpolicy()` 设置和获取 `attr` 参数中的 `schedpolicy` 属性。

`policy` 的允许值包括 `SCHED_FIFO`、`SCHED_RR` 或 `SCHED_OTHER`，它们将由头 `<pthread.h>` 定义。`policy` 值的意义与 § 13.2 中定义的 `schedpolicy` 的值意义相同。当通过规划策略 `SCHED_FIFO` 或 `SCHED_RR` 执行的线程等待着某互斥量时，如果互斥量取消了锁定，它们将按优先权顺序获利互斥量。参见 § 11.3 了解互斥量处理细节。

否则：

或者实现支持 `pthread_attr_setscope()`、`pthread_attr_getscope()`、`pthread_attr_setinheritsched()`、`pthread_attr_getinheritsched()`、`pthread_attr_setschedpolicy()` 和 `pthread_attr_getschedpolicy()` 函数（如以上所描述），或者不提供 `pthread_attr_setscope()`、`pthread_attr_getscope()`、`pthread_attr_setinheritsched()`、`pthread_attr_getinheritsched()`、`pthread_attr_setschedpolicy()` 和 `pthread_attr_getschedpolicy()` 函数。

如果定义了 `{_POSIX_THREADS}`：

函数 `pthread_attr_setschedparam()` 和 `pthread_attr_getschedparam()` 设置和获取 `attr` 参数中的 `schedparam` 参数。§ 13.1 定义了 `param` 结构的内容。对于 `SCHED_FIFO` 和 `SCHED_RR` 策略，`param` 的唯一要求成员是 `sched_priority`。

否则：

或者实现支持 `pthread_attr_setschedparam()` 和 `pthread_attr_getschedparam()` 函数（如上所描述），或者不提供 `pthread_attr_setschedparam()` 和 `pthread_attr_getschedparam()` 函数。

§ 13.5.1.3 返回值

如果成功，`pthread_attr_setscope()`、`pthread_attr_getscope()`、`pthread_attr_setinheritsched()`、`pthread_attr_getinheritsched()`、`pthread_attr_setschedpolicy()`、`pthread_attr_getschedpolicy()`、`pthread_attr_setschedparam()` 和 `pthread_attr_getschedparam()` 函数将返回 0。否则，返回一个表征该错误的错误号。

§ 13.5.1.4 错误

如果发生下面条件之一，`pthread_attr_setscope()`、`pthread_attr_getscope()`、

`pthread_attr_setinheritsched()`、`pthread_attr_getinheritsched()`、`pthread_attr_setschedpolicy()`、`pthread_attr_getschedpolicy()`、`pthread_attr_setschedparam()`和`pthread_attr_getschedparam()`函数将返回对应的错误号：

[ENOSYS] 实现不支持`pthread_attr_setscope()`、`pthread_attr_getscope()`、`pthread_attr_setinheritsched()`、`pthread_attr_getinheritsched()`、`pthread_attr_setschedpolicy()`、`pthread_attr_getschedpolicy()`、`pthread_attr_setschedparam()`和`pthread_attr_getschedparam()`函数。

如果满足以下条件之一，`pthread_attr_setscope()`、`pthread_attr_setinheritsched()`、`pthread_attr_setschedpolicy()`和`pthread_attr_setschedparam()`函数将返回对应的错误号：

[EINVAL]被设置的属性值无效。

[ENOTSUP] 试图设置属性的值为非支持值。

§ 13.5.1.5 交叉参考

`pthread_attr_init()`，§ 16.2.1；`pthread_create()`，§ 16.2.2。

§ 13.5.2 动态线程规划参数访问

函数：`pthread_getschedparam()`、`pthread_setschedparam()`

§ 13.5.2.1 纲要

```
#include <pthread.h>
int pthread_getschedparam(pthread_t thread, int *policy,
                          struct pthread_param *param);
int pthread_setschedparam(pthread_t thread, int policy,
                          const struct pthread_param *param);
```

§ 13.5.2.2 描述

如果定义了`{_POSIX_THREAD_PRIORITY_SCHEDULING}`：

`pthread_getschedparam()`和`pthread_setschedparam()`函数允许读取和设置多线程进程中单个线程的规划策略和规划参数。对于`SCHED_FIFO`和`SCHED_RR`，`sched_param`结构唯一要求的成员是优先权`sched_priority`。对于`SCHED_OTHER`，受影响的规划参数由实现定义。

`pthread_getschedparam()`函数读取由`thread`给定线程ID的线程的规划策略和规划参数，并分别在`policy`和`param`中保存这些值。从`pthread_getschedparam()`返回的优先权值将是影响目标线程的最近`pthread_setschedparam()`或`pthread_create()`调用指定的值。它不反映任何优先权继承或顶置函数对优先权产生的任何临时调整。对于线程ID由`thread`给出的线程，`pthread_setschedparam()`函数将其规划策略和相关规划参数分别设置成`policy`和`param`提供的策略和相关参数。

`policy`参数可能具有值`SCHED_OTHER`、`SCHED_FIFO`或`SCHED_RR`。`SCHED_OTHER`策略的规划参数由实现定义。`SCHED_FIFO`和`SCHED_RR`策略有单一规划参数`sched_priority`。

如果`pthread_setschedparam()`失败，目标线程的规划参数不会改变。

否则：

或者实现支持`pthread_getschedparam()`和`pthread_setschedparam()`函数（如上所描述），或者不提供`pthread_getschedparam()`和`pthread_setschedparam()`函数。

§ 13.5.2.3 返回值

如果成功, `pthread_attr_getscope()`和 `pthread_attr_setscope()`函数将返回 0。否则, 返回一个表征该错误的错误号。

§ 13.5.2.4 错误

如果发生下面条件之一, `pthread_getschedparam()`和 `pthread_setschedparam()`函数将返回对应的错误号:

[ENOSYS] 没有定义选项{`_POSIX_THREAD_PRIORITY_SCHEDULING`}; 而且实现不支持 `pthread_getschedparam()`和 `pthread_setschedparam()`函数。

如果满足以下条件之一, `pthread_getschedparam()`函数将返回对应的错误号:

[ESRCH] `thread` 指定的值没有指向一个存在的线程。

如果满足以下条件之一, `pthread_setschedparam()`函数将返回对应的错误号:

[EINVAL] `policy` 指定的值或与规划策略 `policy` 相关的规划参数之一无效。

[ENOTSUP] 试图将策略或规划参数设置成一个不支持值。

[EPERM] 调用者没有设置指定线程规划参数或规划策略的相应权限。

实现不允许应用程序将其中的参数修改成指定值。

[ESRCH] `thread` 指定的值不是指向一个存在的线程。

§ 13.5.2.5 交叉参考

`sched_setparam()`, § 13.3.1; `sched_getparam()`, § 13.3.2; `sched_setscheduler()`, § 13.3.3; `sched_getscheduler()`, § 13.3.4。

§ 13.6 同步规划

§ 13.6.1 互斥量初始化规划属性

函数: `pthread_mutexattr_setprotocol()`、`pthread_mutexattr_getprotocol()`、`pthread_mutexattr_setprioceiling()`、`pthread_mutexattr_getprioceiling()`

§ 13.6.1.1 纲要

```
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);
int pthread_mutexattr_getprotocol(
    const pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_setprioceiling(
    pthread_mutexattr_t *attr, int prioceiling);
int pthread_mutexattr_getprioceiling(
    const pthread_mutexattr_t *attr, int *prioceiling);
```

§ 13.6.1.2 描述

如果至少定义了{`_POSIX_THREAD_PRIO_INHERIT`}或{`_POSIX_THREAD_PRIO_PROTECT`}其中之一:

这些函数将操纵 `attr` 指向的互斥量属性对象, `attr` 已经在前面由函数 `pthread_mutexattr_init()`

创建（参见 § 11.3.1），

`pthread_mutexattr_t` 互斥量属性对象至少包含 `protocol` 属性。

如果定义了符号 `{_POSIX_THREAD_PRIO_PROTECT}`，`pthread_mutexattr_t` 互斥量属性对象包含 `prioceiling` 属性。

`prioceiling` 属性包含已初始互斥量的优先权顶置。`prioceiling` 的值应位于 `SCHED_FIFO` 定义的优先权最大范围之内。

`protocol` 属性定义使用互斥量时遵循的协议。`protocol` 的值可以为 `PTHREAD_PRIO_NONE`、`PTHREAD_PRIO_INHERIT` 或 `PTHREAD_PRIO_PROTECT` 之一，它们由头 `<pthread.h>` 定义。如果定义了符号 `{_POSIX_THREAD_PRIO_PROTECT}`，`PTHREAD_PRIO_PROTECT` 值有效，如果定义了符号 `{_POSIX_THREAD_PRIO_INHERIT}`，`PTHREAD_PRIO_INHERIT` 值有效。

当某线程通过 `PTHREAD_PRIO_NONE` 协议属性占有一个互斥量时，它的优先权和规划不受它的互斥量占有权的影响。

当某线程由于通过 `PTHREAD_PRIO_INHERIT` 协议属性占有一个或多个互斥量正阻塞较高优先权线程时，它在它自己的优先权之上执行，或者在等待着该线程占有且通过协议初始化的任何互斥量的最高优先权线程执行。

当某个线程占有一个或多个通过 `PTHREAD_PRIO_PROTECT` 协议初始化的互斥量时，它或者在它自己的优先权之上执行，或者在该线程占有且通过此属性初始化的所有互斥量优先权顶置执行，而不管其它线程是否在这些互斥量上阻塞。`prioceiling` 属性定义了初始化互斥量的优先权顶置，它是执行互斥量守护的临界区的最小优先级。为了避免优先权倒置，互斥量的优先权顶置应该设置成高于或等于可以阻塞该互斥量的所有线程的最高优先权。`prioceiling` 的值位于 `SCHED_FIFO` 规划策略之下定义的最大优先权范围之内。

当某线程占据一个通过 `PRIO_INHERIT` 或 `PRIO_PROTECT` 协议属性初始化的互斥量时，它不会被移动到它自己优先权规划队列中的尾部（在原始优先权变化的情况下，如对 `sched_setsparam()` 的调用来更改）。同样，当线程取消锁定一个通过 `PRIO_INHERIT` 或 `PRIO_PROTECT` 协议属性初始化的互斥量时，它不会被移到它自己优先权规划队列的尾部（在原始优先权更改的情况下）。

如果线程同时占有多个通过不同协议初始化的互斥量，它将在所有这些协议可以获得的最高优先权进行执行。

如果定义了符号 `{_POSIX_THREAD_PRIO_INHERIT}`，当线程在某互斥量上调用 `pthread_mutex_lock()`，此互斥量通过具有值 `PTHREAD_PRIO_INHERIT` 的协议属性初始化，同时由于另一个线程占有互斥量导致调用线程阻塞，此时，占有者线程只要继续占有该互斥量，它将继承调用线程的优先级。实现将其执行优先权更新为分配优先权和所有继承优先权的最大者。而且，如果占有者线程自身在另一个互斥量上阻塞，相同的优先权继承影响将以递归方式传播到其它占有者线程。

否则：

或者实现支持 `pthread_mutexattr_setprotocol()`、`pthread_mutexattr_getprotocol()`、`pthread_mutexattr_setprioceiling()` 和 `pthread_mutexattr_getprioceiling()` 函数（如上所描述），或者没有提供 `pthread_mutexattr_setprotocol()`、`pthread_mutexattr_getprotocol()`、`pthread_mutexattr_setprioceiling()`

和 `pthread_mutexattr_getprioceiling()` 函数。

§ 13.6.1.3 返回值

当成功完成时，`pthread_mutexattr_setprotocol()`、`pthread_mutexattr_getprotocol()`、`pthread_mutexattr_setprioceiling()`和`pthread_mutexattr_getprioceiling()`函数将返回 0。否则，将返回一个表征该错误的错误号。

§ 13.6.1.4 错误

如果发生以下条件之一，`pthread_mutexattr_setprotocol()`、`pthread_mutexattr_getprotocol()`、`pthread_mutexattr_setprioceiling()`和`pthread_mutexattr_getprioceiling()`函数将返回相应的错误号：

如果发生下面条件之一，`pthread_mutexattr_setprotocol()`、`pthread_mutexattr_getprotocol()`、`pthread_mutexattr_setprioceiling()`和`pthread_mutexattr_getprioceiling()`函数将返回对应的错误号：

[ENOSYS] 没有定义选项 `{_POSIX_THREAD_PRIO_PROTECT}`，而且实现不支持 `pthread_mutexattr_setprioceiling()`和`pthread_mutexattr_getprioceiling()`函数。

[ENOTSUP] `protocol` 指定的值是一个不支持值。

如果满足以下条件之一，`pthread_mutexattr_setprotocol()`、`pthread_mutexattr_getprotocol()`、`pthread_mutexattr_setprioceiling()`和`pthread_mutexattr_getprioceiling()`函数将返回对应的错误号：

[EINVAL] `attr`、`protocol` 或 `prioceiling` 指定的值无效。

[EPERM] 调用者没有执行操作的权限。

§ 13.6.1.5 交叉参考

`pthread_create()`，§ 16.2；`pthread_mutex_init()`，§ 11.3.2；`pthread_cond_init()`，§ 11.4.2。

§ 13.6.2 更改互斥量优先权顶置

函数：`pthread_mutex_getprioceiling()`，`pthread_mutex_setprioceiling()`

§ 13.6.2.1 纲要

```
#include <pthread.h>

int pthread_mutex_setprioceiling(pthread_mutex_t *mutex,
                                int prioceiling, int *old_ceiling);
int pthread_mutex_getprioceiling(const pthread_mutex_t *mutex,
                                int *prioceiling);
```

§ 13.6.2.2 描述

如果定义了 `{_POSIX_THREAD_PRIO_PROTECT}`：

`pthread_mutex_getprioceiling()`函数返回互斥量的当前优先权顶置。

`pthread_mutex_setprioceiling()`函数在互斥量未锁定时锁定它，或者一直阻塞到可以成功锁定互斥量为止；然后更改互斥量优先权顶置并释放这个互斥量。若更改成功，在 `old_ceiling` 中返回优先权顶置的前一个值。锁定互斥量的进程不需要遵循优先权保护协议。

如果 `pthread_mutex_setprioceiling()`函数失败，互斥量优先权顶置不会改变。

否则：

或者实现支持 `pthread_mutex_getprioceiling()`和`pthread_mutex_setprioceiling()`函数（如上所描述），或者没有提供 `pthread_mutex_getprioceiling()`和`pthread_mutex_setprioceiling()`函数。

§ 13.6.2.3 返回值

如果成功 `pthread_mutex_setprioceiling()`和 `pthread_mutex_getprioceiling()`函数返回 0。否则，返回一个表征该错误的错误号。

§ 13.6.2.4 错误

如果发生下面条件之一，`pthread_mutex_getprioceiling()`和 `pthread_mutex_setprioceiling()`函数将返回对应的错误号：

[ENOSYS] 没有定义选项 `{_POSIX_THREAD_PRIO_PROTECT}`，而且实现不支持 `pthread_mutex_getprioceiling()`和 `pthread_mutex_setprioceiling()`函数。

如果满足以下条件之一，`pthread_mutex_getprioceiling()`和 `pthread_mutex_setprioceiling()`函数将返回对应的错误号：

[EINVAL]prioceiling 指定的优先权越界。

mutex 指定的值不是指向一个当前存在的互斥量。

[ENOSYS] 实现不支持互斥量的优先权顶置协议。

[EPERM] 调用者没有执行操作的权限。

§ 13.6.2.5 交叉参考

`pthread_mutex_init()`, § 11.3.2; `pthread_mutex_lock()`, § 11.3.3; `pthread_mutex_unlock()`, § 11.3.3; `pthread_mutex_trylock`, § 11.3.3。

第 14 节 时钟与定时器

§ 14.1 时钟与定时器的数据定义

头文件 `<time.h>` 定义了定时工具使用的类型和表示常量。

§ 14.1.1 时间值定义结构

许多定时工具函数接受或返回时间值定义。时间值结构 `timespec` 指定一个时间值，并至少包含表 3 所示的成员。

表 3 `timespec` 包含成员

成员类型	成员名字	描述
<code>time_t</code>	<code>tv_sec</code>	秒
<code>long</code>	<code>tv_nsec</code>	纳秒

实现可以按 § 1.3.1.1，第(2)款的规定添加扩展。为此结构添加扩展时，若结构中的这些字段未初始化时可能会更改与标准相关的应用程序的行为，同时要求这种扩展与 § 1.3.1.1 的要求一致。

`tv_nsec` 成员仅在大于或等于 0，并且小于 10^9 纳秒（即 1 秒）时有效。这个结构描述的时间段为 $(tv_sec \times 10^9 + tv_nsec)$ 纳秒。

时间值结构 `itimerspec` 定义了一个初始定时器值以及每个进程定时器函数使用的重复时间段。这个结构至少包含表 4 所示的成员。

表 4 `itimerspec` 包含成员

成员类型	成员名字	描 述
<code>struct timespec</code>	<code>it_interval</code>	定时器周期
<code>struct timespec</code>	<code>it_value</code>	定时器截止时间

实现可以按 § 1.3.1.1, 第(2)款的规定添加扩展。为此结构添加扩展时, 若结构中的这些字段未初始化时可能会更改与标准相关的应用程序的行为, 同时要求这种扩展与 § 1.3.1.1 的要求一致。

如果 `it_value` 的值中为非零, 它表示时间或下一个定时器过期的时间 (分别对应于相对和绝对定时器时间)。如果 `it_value` 描述的值为 0, 则取消了定时器。

如果 `it_interval` 描述的值为非零, 它指定定时器过期后重新装载定时器所用的时间段——也就是说, 指定一个周期性的定时器。如果 `it_interval` 描述的值为零, 定时器下一次过期后将被取消——也就是说, 指定了一个“一次性”的定时器。

实现可以按 § 1.3.1.1, 第(2)款的规定为这些结构添加扩展。为这些结构添加扩展时, 若结构中的这些字段未初始化时, 可能会更改与标准相关的应用程序的行为, 同时要求这种扩展与 § 1.3.1.1 的一致。

§ 14.1.2 定时器事件通知控制块

对于支持实时信号扩展 (Realtime Signals Extension) 选项的实现, 可以为每个进程创建定时器, 通过实现时扩展信号通知定时器过期的进程。在 `<signal.h>` 中定义的 `sigevent` 结构用于创建这种定时器。`sigevent` 结构包含通知调用进程定时器过期事件所使用的信号号 (signal number) 以及特定应用程序数据值。

§ 14.1.3 类型定义

表 5 中的类型由实现在 `<sys/types.h>` 中定义。

表 5 `<sys/types.h>` 中定义的类型

定义类型	描 述
<code>clockid_t</code>	在时钟和定时器函数中用为时钟 ID 类型
<code>timer_t</code>	用作 <code>timer_create()</code> 返回的定时器 ID

§ 14.1.4 显示常量

下面的常量在 `<time.h>` 中定义:

`CLOCK_REALTIME`

系统范围内实时时钟的标识符。

`TIMER_ABSTIME`

表明时间为“绝对”的标志，它针对与某定时器相关的时钟。

`CLOCK_REALTIME` 时钟以及基于该时钟的所有定时器，包括 `nanosleep()` 函数的最大允许分辨率都由 `{_POSIX_CLOCKRES_MIN}` 表示，而且定义为 20 毫秒（即 1/50 秒）。实现可以支持 `CLOCK_REALTIME` 时钟的更小分辨率值，以提供更精细的时间基础。对于特定时钟，实现所运行的实际分辨率使用本章中定义的函数来获得。对于基于时钟的定时器的 `nanosleep()` 函数的实际分辨率不同于该时钟支持的分辨率，则实现要诠释这种差异。

对于 `CLOCK_REALTIME` 时钟以及基于它的分辨率定时器与 C 标准(2)针对 `time_t` 类型定义的一样。如果 `nanosleep()` 函数或基于该时钟定时器支持的最大值与该时钟支持的最大值不同，则实现要诠释这种差异。

§ 14.2 时钟与定时器函数

§ 14.2.1 时钟

函数: `clock_settime()`, `clock_gettime()`, `clock_getres()`

§ 14.2.1.1 纲要

```
#include <time.h>
int clock_settime(clockid_t clock_id, const struct timespec *tp);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_getres(clockid_t clock_id, struct timespec *res);
```

§ 14.2.1.2 描述

如果定义了 `{_POSIX_TIMERS}`:

`clock_settime()` 函数将设置指定时钟 `clock_id` 为 `tp` 指定的值。将位于指定时钟分辨率两个连续非负整数倍数之间的时间值截短为分辨率的较小倍数。

`clock_gettime()` 函数返回指定时钟 `clock_id` 的当前 `tp` 值。

调用 `clock_getres()` 可以获得任何时钟的分辨率。时钟分辨率通过实现定义，不能由进程设置。如果参数 `res` 不是 `NULL`，则指定时钟的分辨率保存到 `res` 指向的位置。如果 `res` 为 `NULL`，则不返回时钟分辨率。如果 `clock_settime()` 的时间参数不是 `res` 的倍数，则将此值截短为 `res` 的倍数。

时钟可以是全系统范围的——即对于所有进程可见；或者是每个进程范围的——记量时间仅在某进程内有意义。所有的实现支持 § 14.1 中定义的 `CLOCK_REALTIME` 的 `clock_id`。这个时钟表示该系统的实时时钟。对于这个时钟，`clock_gettime()` 返回的值以及 `clock_settime()` 指定的值表示从新纪元开始的时间量（以秒和纳秒为单位）。实现也可以支持其它时钟。对这些时钟的时间值的解释未定义。

通过 `clock_settime()` 对每个与该时钟相关进程定时器设置时钟的影响由实现定义。

设置特定时钟的适当权限由实现定义。

否则：

或者实现支持 `clock_settime()`、`clock_gettime()` 和 `clock_getres()` 函数（如上所描述），或者 `clock_settime()`、`clock_gettime()` 和 `clock_getres()` 函数失败。

§ 14.2.1.3 返回值

返回值为 0 表示调用成功。返回值为 -1 表示发生错误，同时设置 `errno` 表征此错误。

§ 14.2.1.4 错误

如果满足条件之一，`clock_settime()`、`clock_gettime()` 和 `clock_getres()` 函数将返回 -1，并将 `errno` 设置成相应的值：

[EINVAL] `clock_id` 参数没有指定一个已知时钟。

[ENOSYS] 实现不支持 `clock_settime()`、`clock_gettime()` 和 `clock_getres()` 函数。

如果满足以下条件之一，`clock_settime()` 函数将返回 -1，并将 `errno` 设置成相应的值：

[EINVAL] `clock_settime()` 的参数 `tp` 位于已知时钟 `id` 的范围之外。

`tp` 参数定义一个小于 0 或大于等于 10^9 的纳秒值。

如果满足以下条件之一，`clock_settime()` 函数将返回 -1，并将 `errno` 设置成相应的值：

[EPERM] 请求进程没有设置指定时钟的适当权限。

§ 14.2.1.5 交叉参考

`timer_gettime()`，§ 14.2.4；`time()`，§ 4.5.1；`ctime()`，§ 8.1。

§ 14.2.2 创建一个各进程定时器

函数：`timer_create()`

§ 14.2.2.1 纲要

```
#include <signal.h>
```

```
#include <time.h>
```

```
int timer_create(clockid_t clock_id, struct sigevent *evp,
                 timer_t *timerid);
```

§ 14.2.2.2 描述

如果定义了 `{_POSIX_TIMERS}`：

`timer_create()` 函数将使用指定时钟 `clock_id` 创建一个各进程定时器作为定时的基础。

`timer_create()` 函数在 `timerid` 引用的位置返回一个 `timer_t` 类型的定时器 ID，`timer_t` 类型用来标识定时器请求中的定时器（参见 § 14.2.4）。删除定时器之前，这个定时器 ID 在调用进程中是唯一的。特定的时钟 `clock_id` 在 `<time.h>` 中定义。被返回 ID 的定时器在从 `timer_create()` 返回时处于取消状态。

如果 `evp` 参数非 NULL，它指向一个 `sigevent` 结构。由应用程序分配的这个结构决定定时器过期时发生的异步通告（在 § 3.3.1.2 中定义）。如果 `evp` 参数为 NULL，它的作用效果就好像 `evp` 参数指向一个 `sigevent` 结构，其中 `sigev_notify` 成员具有值 `SIGEV_SIGNAL`，`sigev_signo` 具有一个缺省信号号，`sigev_value` 成员具有定时器 ID 的值。

每个实现都将定义一套可以用作各进程定时器定时基础的时钟。所有的实现都支持 `CLOCK_REALTIME` 的一个 `clock_id`。

各进程定时器不会被子进程越过 `fork()` 继承，而且被一个 `exec` 取消和删除。

否则：

或者实现支持 `timer_create()` 函数（如上所描述），或者 `timer_create()` 函数失败。

§ 14.2.2.3 返回值

如果调用成功，`timer_create()` 将返回 0，并将 `timerid` 指向的位置更新为 `timer_t`，它可以传递给各进程定时器调用（参见 § 14.2.4）。如果发生错误，函数将返回值 -1，并设置 `errno` 表征此错误。如果发生错误，`timerid` 的值未定义。

§ 14.2.2.4 错误

如果发生以下条件之一，`timer_create()` 函数将返回 -1，并将 `errno` 设置成相应值：

[EAGAIN] 系统缺乏足够的信号队列资源来满足请求。

调用进程已经通过实现创建了允许它创建的所有定时器。

[EINVAL] 没有定义指定时钟 ID。

[ENOSYS] 实现不支持函数 `timer_create()`。

§ 14.2.2.5 交叉参考

<time.h>, § 14.1; `timer_delete()`, § 14.2.3; `clock_gettime()`, § 14.2.1; `clock_settime()`, § 14.2.1; `clock_getres()`, § 14.2.1; `timer_gettime()`, § 14.2.4; `timer_settime()`, § 14.2.4。

§ 14.2.3 删除各进程定时器

函数：`timer_delete()`

§ 14.2.3.1 纲要

```
#include <time.h>
```

```
int timer_delete(timer_t timerid);
```

§ 14.2.3.2 描述

如果定义了 `{_POSIX_TIMERS}`：

`timer_delete()` 函数删除指定定时器 `timerid`，它是由 `timer_create()` 函数在以前创建的。如果调用 `timer_delete()` 时配备了定时器，其行为就像定时器在删除前自动取消一样。为删除定时器部署待决信号未定义。

否则：

或者实现支持 `timer_delete()` 函数（如上所描述），或者 `timer_delete()` 函数失败。

§ 14.2.3.3 返回值

如果成功，函数将返回值 0。否则，函数将返回值 -1，并设置 `errno` 表征该错误。

§ 14.2.3.4 错误

如果发生下面条件之一，`timer_delete()` 函数将返回 -1，并将 `errno` 设置成相应值：

[EINVAL] `timerid` 指定的定时器 ID 不是一个有效定时器 ID。

[ENOSYS] 实现不支持函数 `timer_delete()`。

§ 14.2.3.5 交叉参考

`timer_create()`, § 14.2.2。

§ 14.2.4 各进程定时器

函数: `timer_settime()`, `timer_gettime()`, `timer_getoverrun()`

§ 14.2.4.1 纲要

```
#include <time.h>
int timer_settime(timer_t timerid, int flags,
    const struct itimerspec *value, struct itimerspec *ovalue);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_getoverrun(timer_t timerid);
```

§ 14.2.4.2 描述

如果定义了 `{_POSIX_TIMERS}`:

`timerid` 从 `value` 参数的成员 `it_value` 中指定的定时器下一次过期前, `timer_settime()` 函数将设置时间, 而且如果 `value` 的 `it_value` 成员为非零时配备定时器。如果调用 `timer_settime()` 时指定定时器已经配备, 调用将重新设置时间, 直到指定了下一个过期的 `value`。如果 `value` 的 `it_value` 成员为 0, 则取消定时器。取消或重新设置定时器对待决过期通告的作用未定义。

如果在参数 `flags` 中没有设置标志 `TIMER_ABSTIME`, `timer_settime()` 的行为就像在下次过期前的时间设置成等于 `value` 的 `it_value` 成员指定的时间段一样。也就是说, 定义器在从发出调用开始 `it_value` 纳秒后过期 (参见 § 14.1.1)。如果在参数 `flags` 中设置标志了 `TIMER_ABSTIME`, `timer_settime()` 的行为就像在下次过期前的时间设置成等于 `value` 的 `it_value` 成员指定的绝对时间与 `timerid` 相关的当前时钟值之差一样。也就是说, 定时器在时钟到达 `value` 的 `it_value` 成员指定的值时过期。如果超过了指定时间, 函数将完成, 并发出过期通知。

将定时器重新装载的值设置成 `value` 的 `it_interval` 成员指定的值。当定时器具备一个非零 `it_interval`, 则指定一个周期性 (或重复性) 定时器。

位于两个连续指定定时器分辨率非负整数倍数之间的定时器值将被截短为该分辨率的较大倍数。量化误差 (quantization error) 不会导致定时器早于截短时间值过期。

如果参数值不是 `NULL`, 函数 `timer_settime()` 将在 `ovalue` 引用的位置保存一个值, 这个值表示定时器过期前的时间段 (若定时器取消, 则这个值为 0), 同时保存前一个定时器重新装载值。`ovalue` 的成员服从定时器的分辨率, 而且它们与此时此刻 `timer_gettime()` 调用返回的值相同。`timer_gettime()` 函数在 `value` 参数指向的空间保存, 直到指定定时器 `timerid` 过期的时间量以及定时器的重新装载值。这个结构的 `it_value` 成员包含定时器过期前的时间量, 如果定时器取消则为 0。这个值作为定时器过期前的时间段返回, 即使定时器提供了绝对时间。`value` 的 `it_interval` 成员包含 `timer_settime()` 最后设置的重新装载值。

在任何时间点, 对于某给定定时器, 只有一个信号排队到进程。当定时器的信号仍然悬而未过期时, 不再有信号排队, 此时发生定时器超时运行 (overrun)。当定时器过期信号提交给某进程, 或由进程接收时, 若实现支持实时信号扩展 (Realtime Signals Extension), 则 `timer_getoverrun()` 函数返回针对定时器的定时器过期超时运行量。超时运行量包含额外的定时过期数, 它发生在信号产生 (排队) 之时与提交或接收之时的中间, 它可以达到但不会包含实现定义的 `{DELAYTIMER_MAX}` 最大值。如果此额外过期数大于或等于 `{DELAYTIMER_MAX}`, 则将超时运行量设置为 `{DELAYTIMER_MAX}`。`timer_getoverrun()` 返回的值应用于定时器最近过期信号的

提交或接收中。如果对于该定时器，没有过期信号提交或接收，或者不支持实时信号扩展，则返回超时运行量的意义未定义。

否则：

或者实现支持 `timer_settime()`、`timer_gettime()`和 `timer_getoverrun()`函数（如上所描述），或者 `timer_settime()`、`timer_gettime()`和 `timer_getoverrun()`函数失败。

§ 14.2.4.3 返回值

如果 `clock_settime()`或 `clock_gettime()`函数成功，返回值 0。如果以上某个函数发生错误，则返回值-1，并设置 `errno` 表征该错误。如果 `timer_getoverrun()`函数成功，它返回 § 14.2.4.2 解释的定时器过期超时运行量。

§ 14.2.4.4 错误

如果发生如下条件之一，`timer_settime()`、`timer_gettime()`和 `timer_getoverrun()`函数将返回-1，并将 `errno` 设置成相应值：

[EINVAL] `timerid` 参数与 `timer_create()`返回的 `id` 不相适应，但却没有被 `timer_delete()`删除。

[ENOSYS] 实现不支持 `timer_settime()`、`timer_gettime()`和 `timer_getoverrun()`函数。

如果满足以下条件之一，`timer_settime()`函数将返回-1，将 `errno` 设置成相应的值：

[EINVAL] `value` 结构指定的纳秒值小于 0，或者大于或等于 10^9 。

§ 14.2.4.5 交叉参考

`clock_gettime()`， § 14.2.1； `timer_create()`， § 14.2.2。

§ 14.2.5 高分辨率睡眠 (High Resolution Sleep)

函数：`nanosleep()`

§ 14.2.5.1 纲要

```
#include <time.h>
```

```
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

§ 14.2.5.2 描述

如果定义了 `{_POSIX_TIMERS}`：

`nanosleep()`函数将导致当前线程从执行中挂起，直到经过了 `rqtp` 参数定义的时间段，将信号提交给了调用线程，而且信号的作用是调用信号捕获函数，或者直到进程终止。挂起时间可能比请求时间长，因为参数值被截短为睡眠分辨率的整数倍，或者因为系统其它行动的规划所致。但是，除了被信号打断的情况外，挂起时间不应该少于 `rqtp` 指定的时间，它是由系统时钟 `CLOCK_REALTIME` 计量的。

使用 `nanosleep()`函数不会对任何信号的作用和阻塞有所影响。

否则：

或者实现支持 `nanosleep()`函数（如上所描述），或者 `nanosleep()`函数失败。

§ 14.2.5.3 返回值

如果由于经过了请求时间 `nanosleep()`函数返回，它的返回值将为 0。

如果由于被某个信号打断而返回，`nanosleep()`函数将返回值-1，将设置 `errno` 表征此中断。如

果 `rmt` 参数为非 `NULL`，它所引用的 `timespec` 结构将被更新为时间段中的剩余时间量（请求时间减去实际睡眠的时间）。如果 `rmt` 参数为 `NULL`，则不返回剩余时间。

如果 `nanosleep()` 函数失败，它将返回值 -1，并设置 `errno` 表征此错误。

§ 14.2.5.4 错误

如果发生以下条件之一，`nanosleep()` 函数将返回 -1，并将 `errno` 设置成相应值：

[EINTR] `nanosleep()` 函数被某个信号打断。

[EINVAL] `rtp` 参数指定一个小于 0 或大于等于 10^9 纳秒的值。

[ENOSYS] 实现不支持 `nanosleep()` 函数。

§ 14.2.5.5 交叉参考

`sleep()`，§ 3.4.3。

第 16 节 线程管理

§ 16.1 线程

本节描述通过本标准可以实现的与多个线程控制有关的功能。

线程是一个进程中的单流程控制。本节定义了一套允许在单个进程内创建和管理多个线程的操作。

尽管实现可以拥有在系统中唯一的线程 ID，但应用程序只是假定线程 ID 在单个进程中是可用而且唯一的。未定义调用 ISO/IEC 9945 本部分定义的任何函数的作用，也未定义将线程的 ID 作为参数从另一个进程进行传递。线程终止后，相应的实现可以自由重用线程 ID，条件是创建线程时将 `detachstate` 属性设置成 `PTHREAD_CREATE_DETACHED`，或者对此线程调用 `pthread_detach()` 或 `pthread_join()`。如果线程分离了，它的线程 ID 就不能有效作为调用 `pthread_detach()` 或 `pthread_join()` 的参数。

§ 16.2 线程函数

§ 16.2.1 线程创建属性

函数：`pthread_attr_init()`，`pthread_attr_destroy()`，`pthread_attr_setstacksize()`，`pthread_attr_getstacksize()`，`pthread_attr_setstackaddr()`，`pthread_attr_getstackaddr()`，`pthread_attr_setdetachstate()`，`pthread_attr_getdetachstate()`

§ 16.2.1.1 纲要

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setstacksize(pthread_attr_t *attr,
                             size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr,
                             size_t *stacksize);
```

```

int pthread_attr_setstackaddr(pthread_attr_t *attr,
                             void *stackaddr);
int pthread_attr_getstackaddr(const pthread_attr_t *attr,
                             void **stackaddr);
pthread_attr_setdetachstate(pthread_attr_t *attr,
                             int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);

```

§ 16.2.1.2 描述

如果定义了 `{_POSIX_THREADS}`:

函数 `pthread_attr_init()` 对给定实现使用的各个属性通过缺省值初始化线程属性对象 `attr`。

每个实现诠释它使用的各个属性以及它们的缺省值, 除非这些值由本标准定义。

当 `pthread_create()` 使用结果属性对象 (可能通过单个属性值来修改) 时, 它定义创建线程的属性。单个属性对象可以用于对 `pthread_create()` 的多重同时调用中。

`pthread_attr_destroy()` 函数用于销毁线程属性对象。实现可能导致 `pthread_attr_destroy()` 将 `attr` 设置为实现指定的无效值。在销毁属性后使用它的行为未定义。

如果定义了符号 `{_POSIX_THREAD_ATTR_STACKSIZE}`, 对于定义了最小堆栈大小的线程 (以字节为单位), 实现支持其 `stacksize` 属性。函数 `pthread_attr_setstacksize()` 和 `pthread_attr_getstacksize()` 设置和获取 `attr` 对象中的线程创建 `stacksize` 属性。

如果定义了 `{_POSIX_THREAD_ATTR_STACKADDR}`, 则实现支持 `stackaddr` 属性, 它指定用于创建线程的堆栈的存储位置。存储大小至少为 `{PTHREAD_STACK_MIN}`。函数 `pthread_attr_setstackaddr()` 和 `pthread_attr_getstackaddr()` 设置和获取 `attr` 对象中的线程创建 `stackaddr` 属性。

`detachstate` 属性控制线程是否以分离状态创建。如果线程以分离状态创建, 则 `pthread_detach()` 或 `pthread_join()` 函数使用新创建线程的 ID 是错误的。

`pthread_attr_setdetachstate()` 和 `pthread_attr_getdetachstate()` 函数设置和获取 `attr` 对象中的 `detachstate` 属性。`detachstate` 参数引用的位置或被设置为 `PTHREAD_CREATE_DETACHED`, 或被设置为 `PTHREAD_CREATE_JOINABLE`。值 `PTHREAD_CREATE_DETACHED` 将导致所有通过 `attr` 创建的线程处于分离状态, 而且值 `PTHREAD_CREATE_JOINABLE` 将导致所有通过 `attr` 创建的线程处于可联合状态。`detachstate` 属性的缺省值为 `PTHREAD_CREATE_JOINABLE`。

否则:

或者实现支持 `pthread_attr_init()`、`pthread_attr_destroy()`、`pthread_attr_setstacksize()`、`pthread_attr_getstacksize()`、`pthread_attr_setstackaddr()`、`pthread_attr_getstackaddr()`、`pthread_attr_setdetachstate()` 和 `pthread_attr_getdetachstate()` (如上所描述), 或者不提供 `pthread_attr_init()`、`pthread_attr_destroy()`、`pthread_attr_setstacksize()`、`pthread_attr_getstacksize()`、`pthread_attr_setstackaddr()`、`pthread_attr_getstackaddr()`、`pthread_attr_setdetachstate()` 和 `pthread_attr_getdetachstate()` 函数。

§ 16.2.1.3 返回值

成功完成之时, `pthread_attr_init()`、`pthread_attr_destroy()`、`pthread_attr_setstacksize()`、`pthread_attr_getstacksize()`、`pthread_attr_setstackaddr()`、`pthread_attr_getstackaddr()`、`pthread_attr_setdetachstate()`和`pthread_attr_getdetachstate()`函数将返回 0 值。否则, 将返回一个表征错误的错误号。如果函数 `pthread_attr_getstacksize()`成功, 它将在 `stacksize` 中保存 `stacksize` 属性值。如果函数 `pthread_attr_getstackaddr()`成功, 它将在 `stackaddr` 中保存 `stackaddr` 属性值。如果函数 `pthread_attr_getdetachstate()`成功, 它将在 `detachstate` 中保存 `detachstate` 属性值。

§ 16.2.1.4 错误

如果发生以下条件之一, `pthread_attr_init()`函数将返回相应的错误号:

[ENOMEM] 初始化线程属性对象内存不足。

如果发生以下条件之一, 函数 `pthread_attr_getstacksize()`和`pthread_attr_setstacksize()`将返回相应的错误号:

[ENOSYS] 选项 {POSIX_THREAD_ATTR_STACKSIZE} 未定义, 而且不支持线程的 `stacksize` 属性。

如果发生以下条件之一, `pthread_attr_setstacksize()`函数将返回相应的错误号:

[EINVAL] `stacksize` 的值小于 {PTHREAD_STACK_MIN} 或超过了系统制定的极限值。

如果发生以下条件之一, `pthread_attr_getstackaddr()`和`pthread_attr_setstackaddr()`函数将返回相应的错误号:

[ENOSYS] 选项 {POSIX_THREAD_ATTR_STACKADDR} 未定义, 而且不支持线程的 `stackaddr` 属性。

如果发生以下条件之一, `pthread_attr_setdetachstate()`函数将返回相应的错误号:

[EINVAL] `detachstate` 的值无效。

§ 16.2.1.5 交叉参考

`pthread_create()`, § 16.2.2。

§ 16.2.2 线程创建

函数: `pthread_create()`

§ 16.2.2.1 纲要

```
#include < pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void *), void *arg);
```

§ 16.2.2.2 描述

如果定义了 {POSIX_THREADS}:

`pthread_create()`函数用于在一个进程内创建一个新线程, 属性由 `attr` 指定。如果 `attr` 为 NULL, 则使用缺省属性。如果 `attr` 指定的属性在后来被修改, 不会影响线程的属性。成功完成之时, `pthread_create()`函数将在 `thread` 引用的位置保存所创建线程的 ID。

将 `arg` 作为唯一参数执行 `start_routine` 创建线程。如果 `start_routine` 返回, 其作用就仿佛使用 `start_routine` 的返回值作为退出状态隐式调用 `pthread_exit()` 一样。请注意, 最先调用 `main()` 的线程与之不同。当这个线程从 `main()` 返回时, 其作用就像使用 `main()` 的返回值作为退出状态隐式调用

exit()一样。

新线程的信号状态初始化如下：

(1) 信号掩码从创建线程者继承。

(2) 新线程的待定义信号集为空。

如果 pthread_create() 失败，不创建新线程，而且 thread 引用位置的内容未定义。

否则：

或者实现支持 pthread_create() 函数（如上所描述），或者不提供 pthread_create() 函数。

§ 16.2.2.3 返回值

如果成功，pthread_create() 函数将返回 0。否则，将返回一个表征错误的错误号。

§ 16.2.2.4 错误

如果发生以下条件之一，pthread_create() 函数将返回相应的错误号：

[EAGAIN] 系统缺乏创建另一个线程的必需资源，或者超过了系统制定的进程中总线程数极限{PTHREAD_THREADS_MAX}。

[EINVAL] attr 指定的值无效。

§ 16.2.2.5 交叉参考

pthread_exit(), § 16.2.5; pthread_join(), § 16.2.3; fork(), § 3.1.1。

§ 16.2.3 等待线程终止

函数：pthread_join()

§ 16.2.3.1 纲要

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **value_ptr);
```

§ 16.2.3.2 描述

如果定义了{ _POSIX_THREADS }：

pthread_join() 函数挂起调用线程的执行，直到目标线程终止，除非目标线程已经终止。当从一个非 NULL value_ptr 参数的 pthread_join() 成功调用返回时，在 value_ptr 引用位置可以使用终止线程传递给 pthread_exit() 的值。

当 pthread_join() 成功返回，目标线程已经终止。对应于同一个目标线程的多个同时 pthread_join() 调用未定义。如果取消线程调用 pthread_join()，则目标线程不会被分离。

对于已经退出但保持未联合的线程，未指定其是否对{ PTHREAD_THREADS_MAX }不利。

否则：

或者实现支持 pthread_join() 函数（如上所描述），或者不提供 pthread_join() 函数。

§ 16.2.3.3 返回值

如果成功，pthread_join() 函数将返回 0。否则，将返回一个表征错误的错误号。

§ 16.2.3.4 错误

如果发生以下条件之一，pthread_join() 函数将返回对应的错误号：

[EINVAL] 实现发现 thread 指定的值不是指向一个可以联合的线程。

[ESRCH] 没有发现对应于给定线程 ID 指定的线程。

对于以下各条件，如果满足其中某条件，`pthread_join()`函数将返回对应的错误号：

[EDEADLK] 发现死锁，或者 `thread` 的值指定调用线程。

§ 16.2.3.5 交叉参考

`pthread_create()`， § 16.2.2； `wait()`， § 3.2.1。

§ 16.2.4 分离线程

函数：`pthread_detach()`

§ 16.2.4.1 纲要

```
#include <pthread.h>
```

```
int pthread_detach(pthread_t thread);
```

§ 16.2.4.2 描述

如果定义了 `{_POSIX_THREADS}`：

`pthread_detach()`函数用于向实现表示，当线程终止时，它的 `thread` 存储空间可以回收。如果 `thread` 没有终止，`pthread_detach()`将让它终止。未定义对同一个目标线程调用多个 `pthread_detach()` 的效果。

否则：

或者实现支持 `pthread_detach()`函数（如上所描述），或者不提供 `pthread_detach()`函数。

§ 16.2.4.3 返回值

如果调用成功，`pthread_detach()`返回 0。否则，返回一个表征该错误的错误号。

§ 16.2.4.4 错误

如果发生以下条件之一，`pthread_detach()`函数将返回对应的错误号：

[EINVAL] 实现发现 `thread` 指定的值不是指向一个可以联合的线程。

[ESRCH] 没有发现对应于给定线程 ID 指定的线程。

§ 16.2.4.5 交叉参考

`pthread_join()`， § 16.2.3。

§ 16.2.5 线程终止

函数：`pthread_exit()`

§ 16.2.5.1 纲要

```
#include <pthread.h>
```

```
void pthread_exit(void *value_ptr);
```

§ 16.2.5.2 描述

如果定义了 `{_POSIX_THREADS}`：

`pthread_exit()`函数终止调用线程并使值 `value_ptr` 可用于所有与终止线程成功的联合。任何被推入但尚未弹出的取消清理处理器（cancellation cleanup handler）应当按推入的相反顺序弹出，然后再执行。执行完所有的取消清理处理器后，如果线程有一些特定线程数据，则按一种非指定顺

序调用适当的析构函数。线程终止不释放任何应用程序可见进程资源，包括（但不限于）互斥量和文件描述符，它也不执行任何进程级清理动作，包括（但不限于）调用可能存在的 `atexit()` 程序。

当某个线程（此线程非首次调用 `main()` 的线程）从用于创建该线程的启动程序返回时，隐式调用 `pthread_exit()` 函数。函数的返回值当作线程的退出状态。

如果从取消清理处理器或析构函数调用 `pthread_exit()` 函数，而且取消清理处理器或析构函数是作为隐式或显式调用 `pthread_exit()` 的结果，则未定义 `pthread_exit()` 的相关行为。

线程终止后，未定义访问该线程局部（自动）变量的结果。因此对现有线程的局部变量引用不应当用于 `pthread_exit()` 的 `value_ptr` 参数值。

最后一个线程终止后，进程应该以退出状态 0 退出。其行为就像线程终止时，实现通过一个 0 参数调用 `exit()`。

否则：

或者实现支持 `pthread_exit()` 函数（如上所描述），或者不提供 `pthread_exit()` 函数。

§ 16.2.5.3 返回值

`pthread_exit()` 函数不能返回到它的调用者。

§ 16.2.5.4 错误

无。

§ 16.2.5.5 交叉参考

`pthread_create()`, § 16.2.2; `pthread_join()`, § 16.2.3; `exit()`, § 8.1; `_exit()`, § 3.2.2。

§ 16.2.6 获取线程 ID

函数: `pthread_self()`

§ 16.2.6.1 纲要

```
#include <pthread.h>
pthread_t pthread_self(void);
```

§ 16.2.6.2 描述

如果定义了 `{_POSIX_THREADS}`：

`pthread_self()` 函数返回调用线程的线程 ID。

§ 16.2.6.3 返回值

参见 § 16.2.6.2。

§ 16.2.6.4 错误

无。

§ 16.2.6.5 交叉参考

`pthread_create()`, § 16.2.2; `pthread_equal()`, § 16.2.7。

§ 16.2.7 比较线程 ID

函数: `pthread_equal()`

§ 16.2.7.1 纲要

```
#include <pthread.h>
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

§ 16.2.7.2 描述

如果定义了{`_POSIX_THREADS`}:

这个函数比较线程 ID `t1` 和 `t2`。

否则:

或者实现支持 `pthread_equal()` 函数 (如上所描述), 或者不提供 `pthread_equal()` 函数。

§ 16.2.7.3 返回值

如果 `t1` 和 `t2` 不相等, `pthread_equal()` 函数返回非 0 值; 否则, 返回 0。

如果 `t1` 或 `t2` 不是有效线程 ID, 则其行为未定义。

§ 16.2.7.4 错误

无。

§ 16.2.7.5 交叉参考

`pthread_create()`, § 16.2.2; `pthread_self()`, § 16.2.6。

§ 16.2.8 动态包初始化 (Dynamic Package Initialization)

函数: `pthread_once()`

§ 16.2.8.1 纲要

```
#include <pthread.h>
```

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

```
int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
```

§ 16.2.8.2 描述

如果定义了{`_POSIX_THREADS`}:

进程中任何线程通过给定 `once_control` 对 `pthread_once()` 的第一次调用将调用无参数的 `init_routine()`。以后通过相同 `once_control` 对 `pthread_once()` 的调用不会调用 `init_routine()`。从 `pthread_once()` 返回时, 它保证 `init_routine()` 已经完成。`once_control` 参数用于判断关联的初始化程序是否被调用。

函数 `pthread_once()` 不是一个取消点 (cancellation point)。不过, 如果 `init_routine()` 是一个取消点, 而且被取消了, 则它对 `once_control` 的作用就像从未调用过 `pthread_once()` 一样。

常量 `PTHREAD_ONCE_INIT` 由头 `<pthread.h>` 定义。

如果 `once_control` 具有自动存储期限, 或者没有被 `PTHREAD_ONCE_INIT` 初始化, 则 `pthread_once()` 的行为未定义。

否则:

或者实现支持 `pthread_once()` 函数 (如上所描述), 或者不提供 `pthread_once()` 函数。

§ 16.2.8.3 返回值

成功完成之时, `pthread_once()` 将返回 0。否则, 返回一个表征错误的错误号。

§ 16.2.8.4 错误

未定义。

第17节 特定线程数据

本节描述 ISO/IEC9945 部分中与线程和数据关联有关的可用工具。

§ 17.1 特定线程数据函数

§ 17.1.1 特定线程数据键创建

函数: `pthread_key_create()`

§ 17.1.1.1 纲要

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key,
    void (*destructor)(void *));
```

§ 17.1.1.2 描述

如果定义了 `{_POSIX_THREADS}`:

该函数创建一个对进程中所有线程可见的特定线程数据键 (thread-specific data key)。 `pthread_key_create()` 提供的键值为用于定位特定线程数据的不透明对象。虽然不同的线程可能使用相同的键值, 但 `pthread_setspecific()` 对值 (value) 到键 (key) 的绑定是以各线程为基础的, 而且在调用线程的生存期内保持不变。

创建键时, 值 `NULL` 与所有激活线程中的新键关联。创建线程时, 值 `NULL` 与新创建线程中的所有定义键关联。

可用一个可选析构函数与每个键值关联。线程退出时, 如果键值有一个非 `NULL` 析构函数指针, 而且如果线程有一个与该键关联的非 `NULL` 值, 则把当前关联值作为唯一参数来调用指向的函数。如果退出时对于某线程存在多个析构函数, 未指定析构函数调用的顺序。

对于与析构函数关联的所有非 `NULL` 值调用了所有的析构函数后, 如果仍然存在与析构函数关联的非 `NULL` 值, 则重复该过程。对于那些未解决的非 `NULL` 值, 至少完成 `{PTHREAD_DESTRUCTOR_ITERATIONS}` 次析构函数调用迭代后, 如果仍然存在与析构函数关联的非 `NULL` 值, 实现可能停止调用析构函数, 或者继续调用析构函数, 直到不再存在与析构函数关联的非 `NULL` 值, 即使可能导致无限循环也会如此。

否则:

或者实现支持 `pthread_key_create()` 函数 (如上所描述), 或者不提供 `pthread_key_create()` 函数。

§ 17.1.1.3 返回值

如果成功, `pthread_key_create()` 函数将保存在 `*key` 保存新创建的键值, 并返回 0。否则, 返回一个表征错误的错误号。

§ 17.1.1.4 错误

如果发生以下条件之一, `pthread_key_create()` 函数将返回对应的错误号:

[EAGAIN] 系统缺乏创建另一个特定线程数据键的资源, 或者超过了系统制定的每个进程

{PTHREAD_KEYS_MAX}总键数极限值。

[ENOMEM] 创建键内存不足。

§ 17.1.1.5 交叉参考

pthread_getspecific(), § 17.1.2; pthread_setspecific(), § 17.1.2; pthread_key_delete(), § 17.1.3。

§ 17.1.2 特定线程数据管理

函数: pthread_setspecific(), pthread_getspecific()

§ 17.1.2.1 纲要

```
#include <pthread.h>
```

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

```
void *pthread_getspecific(pthread_key_t key);
```

§ 17.1.2.2 描述

如果定义了{ _POSIX_THREADS }:

pthread_setspecific()函数关联特定线程值与通过前一个 pthread_key_create()调用获得的键。不同线程可以在同一个键上绑定不同的值。

这些值一般是指向动态分配内存块的指针, 这些内存块保留供调用线程使用。

pthread_getspecific()函数返回当前绑定到代表调用线程的指定键的值。

通过不是从 pthread_key_create()获取的键值调用 pthread_setspecific()或 pthread_getspecific(), 或者通过 pthread_key_delete()删除键后的效果未定义。

pthread_setspecific()和 pthread_getspecific()可以从特定线程数据析构函数中调用。不过, 从析构函数调用 pthread_setspecific()可能导致丢失存储空间或无限循环。

两个函数都可以实现为宏 (macro)。

否则:

或者实现支持 pthread_setspecific()和 pthread_getspecific()函数 (如上所描述), 或者没有提供 pthread_setspecific()和 pthread_getspecific()函数。

§ 17.1.2.3 返回值

函数 pthread_getspecific()返回与给定 key 相关的特定线程数据值。如果没有与 key 关联的特定线程数据值, 则返回值 NULL。

如果成功, pthread_setspecific()函数将返回 0。否则, 返回一个表征错误的错误号。

§ 17.1.2.4 错误

如果发生以下条件之一, pthread_setspecific()函数将返回对应的错误号:

[ENOMEM] 关联值与键的现有内存不足。

对于以下条件, 如果满足以下条件之一, pthread_setspecific()函数将返回对应的错误号:

[EINVAL] 键值无效。

从 pthread_getspecific()没有返回错误。

§ 17.1.2.5 交叉参考

pthread_key_create(), § 17.1.1。

§ 17.1.3 特定线程数据键删除

函数: `pthread_key_delete()`

§ 17.1.3.1 纲要

```
#include <pthread.h>
```

```
int pthread_key_delete(pthread_key_t key);
```

§ 17.1.3.2 描述

如果定义了 `{_POSIX_THREADS}`:

这个函数删除 `pthread_key_create()` 函数前一次返回的特定线程数据键。与键关联的特定线程数据在调用 `pthread_key_delete()` 时不需要 `NULL`。由应用程序负责释放应用程序存储空间, 或者对任何线程中删除键或关联特定线程数据相关的数据结构执行所有清理动作; 这种清理可以在调用 `pthread_key_delete()` 之前或之后进行。在调用 `pthread_key_delete()` 之后试图使用键将导致未定义的行为。

从析构函数中可以调用 `pthread_key_delete()` 函数。`pthread_key_delete()` 不能调用任何析构函数。任何与 `key` 关联的析构函数将不再在线程退出时被调用。

否则:

或者实现支持 `pthread_key_delete()` 函数 (如上所描述), 或者没有提供 `pthread_key_delete()` 函数。

§ 17.1.3.3 返回值

如果成功, `pthread_key_delete()` 将返回 0。否则, 返回一个表征错误的错误号。

§ 17.1.3.4 错误

对于以下各条件, 如果满足其中一个条件, `pthread_key_delete()` 函数将返回对应的错误号: `[EINVAL]` key 值无效。

§ 17.1.3.5 交叉参考

`pthread_key_create()`, § 17.1.1。

第 18 节 线程取消

本节定义一套允许取消线程的操作。

首先讲解与线程取消绑定的 C 语言, 然后讲解与语言无关的线程取消功能, 它是所有语言绑定所需要的。

§ 18.1 线程取消概述

线程取消机制允许线程按一种受控制方式终止进程中其它线程的执行。允许目标线程 (即被取消的线程) 以多种方式让取消请求待决, 同时, 当它作用于取消通知时, 它也可以执行特定应用程序的清理过程。

取消过程由取消控制接口 (cancellation control interface) 控制。每个线程保持有自己的“取消状态”。取消可以只发生在取消点, 或发生在线程可异步取消之时。

本节描述的线程取消机制依赖于具有延迟取消能力集 (deferred cancelability set) 的程序, 它被指定为缺省情况。应用程序还需要在它们的执行行为中认真遵循静态词法域规则 (static lexical scoping rule)。例如, 使用 `setjmp()`、`return`、`goto` 等让用户自定义取消域不进行必需的域弹出操作将导致未定义行为。

在占据可能需要释放的资源时, 使用异步取消能力可能导致资源丢失。同样, 取消域只在线程处于延迟 (deferred) 或禁止 (disable) 取消状态时才可以被安全操纵。

§ 18.1.1 取消能力状态

线程的取消能力状态 (cancelability state) 决定了对接收的取消请求采取的动作。线程可以通过许多方式来控制取消。

每个线程保持它自己的“取消能力状态”, 它可以用两个位进行编码:

允许取消能力 (cancelability enable): 当取消能力为 `PTHREAD_CANCEL_DISABLE`, 对目标线程的取消请求将保持待决。缺省时, 取消能力设置为 `PTHREAD_CANCEL_ENABLE`。

取消能力类型 (cancelability type): 当允许取消能力, 而且取消能力类型为 `PTHREAD_CANCEL_ASYNCHRONOUS`, 新的或待决取消请求可能在任何时候得到处理。当允许取消能力, 而且取消能力类型为 `PTHREAD_CANCEL_DEFERRED` 时, 取消请求在到达取消点前 (参见下面内容) 一直保持待决。如果禁止取消能力, 取消能力类型的设置没有即时影响, 所有取消请求保持待决; 不过, 一旦再次允许取消能力, 新类型将发挥作用。在所有新创建的线程中, 取消能力都是 `PTHREAD_CANCEL_DEFERRED`, 包括在第一次调用 `main()` 的线程中也是如此。

§ 18.1.2 取消点

取消点 (cancellation point) 发生在线程执行如下 POSIX.1 或 C 标准[2]函数之时:

<code>aio_suspend()</code>	<code>pause()</code>	<code>sigwait()</code>
<code>close()</code>	<code>pthread_cond_timedwait()</code>	<code>sigwaitinfo()</code>
<code>creat()</code>	<code>pthread_cond_wait()</code>	<code>sleep()</code>
<code>fsync()</code>	<code>pthread_join()</code>	<code>system()</code>
<code>mq_receive()</code>	<code>pthread_testcancel()</code>	<code>tcdrain()</code>
<code>mq_send()</code>	<code>read()</code>	<code>wait()</code>
<code>msync()</code>	<code>sem_wait()</code>	<code>waitpid()</code>
<code>nanosleep()</code>	<code>sigsuspend()</code>	<code>write()</code>
<code>open()</code>	<code>sigtimedwait()</code>	
<code>fcntl()</code>	(当 <code>cmd</code> 参数为 <code>F_SETLKW</code> 之时)	

取消点也可以发生在线程执行如下 POSIX.1 或 C 标准[2]函数之时:

<code>closedir()</code>	<code>ftell()</code>	<code>getpwnam()</code>	<code>prts()</code>
<code>ctermid()</code>	<code>fwrite()</code>	<code>getpwnam_r()</code>	<code>readdir()</code>
<code>fclose()</code>	<code>getc()</code>	<code>getpwuid()</code>	<code>remove()</code>
<code>fflush()</code>	<code>getc_unlocked()</code>	<code>getpwuid_r()</code>	<code>rename()</code>
<code>fgetc()</code>	<code>getchar()</code>	<code>gets()</code>	<code>rewind()</code>
<code>fgets()</code>	<code>getchar_unlocked()</code>	<code>lseek()</code>	<code>rewinddir()</code>

fopen()	getcwd()	opendir()	scanf()
fprintf()	getgrgid()	perror()	tmpfile()
fputc()	getgrgid_r()	printf()	tmpname()
fputs()	getgrnam()	putc()	ttyname()
fread()	getgrnam_r()	putc_unlocked()	ttyname_r()
freopen()	getlogin()	putchar()	ungetc()
fscanf()	getlogin_r()	putchar_unlocked()	unlink()
fseek()			
fcntl()	(针对命令参数的任何值)		

实现不会在其它任何 POSIX.1 或 C 标准[2]函数中引入取消点。

调用 POSIX.1 函数期间取消请求挂起，这一作用的副影响是与单线程程序中某函数调用被信号打断，而且给定函数返回[EINTR]时的副作用是相同的。任何此类副作用应该发生在调用任何取消清理处理器之前。

只要线程具备取消能力，而且将此线程作为目标发出了取消请求，同时该线程调用 pthread_testcancel()，取消请求将在 pthread_testcancel() 返回前发挥作用。如果线程具备取消能力，而且线程具有异步取消请求待决，同时该线程在取消点挂起，等待某事件的发生，则取消请求将起作用。不过，如果线程在取消点挂起，而且它所等待的事件在取消请求发挥作用前发生，未定义取消请求发挥作用或请求保持待决且线程恢复正常执行。

§ 18.1.3 线程取消清理处理器

每个线程保持有一个取消清理处理器 (cancellation cleanup handler) 列表。程序员使用 pthread_cleanup_push() 和 pthread_cleanup_pop() 函数将程序 (routine) 放入列表，或从列表删除程序 (routine)。

当取消请求发挥作用时，按 LIFO 序列逐个调用列表中的程序；即最后推入列表中的程序（对应 LI, Last In）首先被调用（对应 FO, First Out）。线程调用取消清理处理器，在最后取消清理处理器返回之前禁止取消。当针对某作用域调用取消清理处理器时，这个作用域的存储空间必须保持有效。如果最后取消清理处理器返回，终止线程执行，而且与目标联合的任何线程都可以使用 PTHREAD_CANCELED 状态。符号常量 PTHREAD_CANCELED 扩展成一个 (void *) 类型的常量表达式，它的值不与内存中的对象指针匹配，也不为值 NULL。

线程调用 pthread_exit() 时也调用取消清理处理器。

在条件变量等待中取消请求发挥作用的副作用是，在调用第一个取消清理处理器前互斥量要重新获取。另外，线程不再看作等待此条件，而且线程不会消费此条件上的任何条件信号。

取消清理处理器不通过 longjmp() 或 siglongjmp() 退出。

§ 18.1.4 异步-取消 (async-cancel) 安全性

函数 pthread_cancel()、pthread_setcancelstate() 以及 pthread_setcanceltype() 为异步取消安全。

在 POSIX.1 或 C 标准[2]中不要求其它函数为异步-取消安全。

§ 18.2 线程取消函数

§ 18.2.1 线程的取消执行

函数: `pthread_cancel()`

§ 18.2.1.1 纲要

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

§ 18.2.1.2 描述

如果定义了 `{_POSIX_THREADS}`:

`pthread_cancel()` 函数请求取消 `thread`。取消能力状态以及目标线程类型决定何时取消发挥作用。当取消发挥作用时, 调用 `thread` 的取消清理处理器。当返回最后取消清理处理器时, 将对 `thread` 调用特定线程数据析构函数。当最后析构函数返回时, 将终止 `thread`。

目标线程中的取消过程与从 `pthread_cancel()` 返回的调用线程异步运行。

否则:

或者实现支持 `pthread_cancel()` 函数 (如上所描述), 或者没有提供 `pthread_cancel()` 函数。

§ 18.2.1.3 返回值

如果成功, `pthread_cancel()` 函数将返回 0。否则, 将返回一个表征错误的错误号。

§ 18.2.1.4 错误

对于以下条件, 若满足某一条件, 则 `pthread_cancel()` 函数返回对应的错误号:

[ESRCH] 没有发现对应于给定线程 ID 指定的线程。

§ 18.2.1.5 交叉参考

`pthread_exit()`, § 16.2.5; `pthread_join()`, § 16.2.3; `pthread_setcancelstate()`, § 18.2.2;
`pthread_cond_wait()`, § 11.4.4; `pthread_cond_timedwait()`, § 11.4.4。

§ 18.2.2 设置取消能力状态

函数: `pthread_setcancelstate()`、`pthread_setcanceltype()`、`pthread_testcancel()`

§ 18.2.2.1 纲要

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

§ 18.2.2.2 描述

如果定义了 `{_POSIX_THREADS}`:

`pthread_setcancelstate()` 函数自动将调用线程的取消能力状态设置成指示的 `state`, 并且返回 `oldstate` 引用位置处的前一个取消能力状态。`state` 的合法值有 `PTHREAD_CANCEL_ENABLE` 和 `PTHREAD_CANCEL_DISABLE`。

`pthread_setcanceltype()` 函数自动将调用线程的取消能力类型设置成指示的 `type`, 并且返回

oldtype 引用位置处的前一个取消能力类型。type 的合法值有 PTHREAD_CANCEL_DEFERRED 和 PTHREAD_CANCEL_ASYNCHRONOUS。

任何新创建线程（包括首先调用 main() 的线程）的取消能力状态和类型分别为 PTHREAD_CANCEL_ENABLE 和 PTHREAD_CANCEL_DEFERRED。

pthread_testcancel() 函数在调用线程中创建了一个取消点。如果禁止了取消能力，则 pthread_testcancel() 无影响。

否则：

或者实现支持 pthread_setcancelstate()、pthread_setcanceltype() 和 pthread_testcancel() 函数（如上所描述），或者没有提供 pthread_setcancelstate()、pthread_setcanceltype() 和 pthread_testcancel() 函数。

§ 18.2.2.3 返回值

如果成功，pthread_setcancelstate() 和 pthread_setcanceltype() 函数将返回 0。否则，将返回一个表征错误的错误号。

§ 18.2.2.4 错误

对于以下各条件，如果满足其中之一，pthread_setcancelstate() 将返回相应的错误号：

[EINVAL] 指定状态不是 PTHREAD_CANCEL_ENABLE 和 PTHREAD_CANCEL_DISABLE。

对于以下各条件，如果满足其中之一，pthread_setcanceltype() 将返回相应的错误号：

[EINVAL] 指定类型不是 PTHREAD_CANCEL_DEFERRED 和 PTHREAD_CANCEL_ASYNCHRONOUS。

§ 18.2.3 建立取消处理器

函数：pthread_cleanup_push(), pthread_cleanup_pop()

§ 18.2.3.1 纲要

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

§ 18.2.3.2 描述

如果定义了 {_POSIX_THREADS}：

pthread_cleanup_push() 函数将指定的取消清理处理器 routine 推入调用线程的取消清理堆栈。线程退出时（即调用 pthread_exit()），取消清理处理器从取消清理堆栈弹出，而且通过参数 arg 被调用，线程取消请求发挥作用，或者线程通过非零 execute 参数调用 pthread_cleanup_pop() 函数。

pthread_cleanup_pop() 函数删除调用线程取消清理堆栈顶层的 routine，并且可选性地调用它（如果 execute 为非零的话）。

在 C 语言中，这些函数可以实现为宏，以语句的形式出现，而且在同一词法域中成对出现（也就是说，可以认为 pthread_cleanup_push() 宏扩展成一个记号列表，列表的第一个记号是 “{”，pthread_cleanup_pop() 宏扩展成最后记号为对应 “}” 的记号列表。

如果由于跳转缓冲器充满，没有匹配性地调用 pthread_cleanup_push() 或

`pthread_cleanup_pop()`，则调用 `longjmp()` 或 `siglongjmp()` 的作用未定义。从取消清理处理器内调用 `longjmp()` 或 `siglongjmp()` 的作用也未定义，除非在取消清理处理器中跳转缓冲器也充满。

否则：

或者实现支持 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数（如上所描述），或者没有提供 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数。

§ 18.2.3.3 返回值

`pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数将用作语句。

§ 18.2.3.4 错误

本标准没有针对 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数定义任何错误条件。

§ 18.2.3.5 交叉参考

`pthread_cancel()`，§ 18.2.1；`pthread_setcancelstate()`，§ 18.2.2。

§ 18.3 语言无关取消功能

本款阐述所有语言绑定所要求的语言无关线程取消功能（language-independent thread cancellation functionality）。

§ 18.3.1 请求取消

对于线程取消的所有语言绑定都将提供一种机制来请求线程当前运行的取消计算。C 语言对这种机制的绑定是 `pthread_cancel()` 函数。

§ 18.3.2 关联清理代码与作用域

所有的线程取消语言绑定都提供一种关联清理代码（它在取消代码作用域时运行）与作用域的机制。这种机制允许建立任意数量的这类作用域以及它们的清理代码，同时让作用域入栈。C 语言对这种机制的绑定为 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数。

§ 18.3.3 在作用域内控制取消

对线程取消的所有语言绑定都将提供一种控制作用域内取消可能发生点的机制。这种机制可以按模块化方式使用，在一个作用域内作出的取消控制选择不会与嵌套或调用作用域作出的选择冲突，或者后者导致前者无效。推荐缺省取消控制环境让取消仅发生定义的取消点。C 语言对这种机制的绑定为 `pthread_setcancelstate()`、`pthread_setcanceltype()` 和 `pthread_testcancel()` 函数。

§ 18.3.4 已定义取消序列

所有线程取消的语言绑定都将定义取消期间出现的执行序列以及执行环境。所有绑定限制于作用域清理代码的执行集，其顺序与关联作用域入栈的顺序相反（即按 LIFO 顺序）。绑定可以为被取消线程内清理代码提供一种机制，表明此取消在所有作用域清理前已经完成，让执行在清理代码关联作用域内恢复。在运行所有清理代码，而且终止被取消线程前，C 语言没有为完成取消提供相应机制。

§ 18.3.5 取消点列表

线程取消的所有语言绑定都将提供一个针对该语言绑定提供的所有函数的完整取消点列表。它的意图是，让这个列表包含那些可能在无限时间内阻塞的函数。ISO/IEC9945 的这一部分提供了该列表。

附录 B 类关系图规范

类关系图是一种源自 Ctest 试验室的设计和文档技术。类关系图可用于显示类之间是否通过继承、包容或其他方法相互关联。类关系图可以显示类是否为单个继承、多个继承或聚集的产物。类关系图可以显示类集合中任何级别的关系细节。对关系的高层次概览可以图表来示意，低层次的关系可以具体化到源代码层次。

类关系图包括 4 个基本组成：

- 关系运算符符号；
- 流程线条；
- 垂直向和水平向定义；
- 类分组符号。

图示技术为面向对象范例引入了原型类（protoclass）抽象。原型类为 n 个类之间的差集。例如，已知两个类 A 和 B，则原型类为 A 与 B 之间的差集。如果类 A 包含成员 {1,2,3,4}，B 包含成员 {1,2,3,4,5}，则 A 与 B 的原型类为 {5}，而且原型类属于类 B。类 A 与类 B 不需要引用不同的类。

每种类关系包括一个空类（null class）。虽然没有在图中显示出来，但提出来是可以理解的。例如，描述单个类与自身之间的关系时，就显示了原型类的存在。已知一个类 A 成员 {1,2,3,4}，求类 A 与空类 {} 之间的差集得到 A 的原型类。因此，{1,2,3,4}/{} 等于 {1,2,3,4}。基类 A 的原型类就是它的成员函数。这表明，每个类都有一个原型类和一个空类。

类关系符号显示类与组件的关联方式。无论类与组件间存在什么关系，都可以在关系符号里表达。流程线条显示表明在某个关系中哪些类与组件组合在一起。流程线条表明两端终点相连的类是关联的。关系符号告知它们的关联方式。对于关系符号中可有哪种关系没有任何限制。所以，CRD（class relationship diagram，类关系图）可以显示类与组件间的任何关系类型。流程线条具有以下语义：A 通过关系符号表明与 B 关联。

水平与垂直向定义（horizontal and vertical specifications）指定在所有面向对象范例中存在的两种基本关系：继承（inheritance）与包容（containment）。如果类或组件通过继承关联，它将用

水平矩形的分组符号来表示。类或组件与派生类有 is-a 关系。如果类或组件通过包容关联，它将通过垂直向矩形分组符号来表示。类或组件与它所包容的类有 has-a 关系。

类分组符号用于显示关系分析中的类或组件集合。类分组符号可以显示类的最高级表示（即类名），或可以显示类的最低级表示（即定义数据成员和成员函数的源代码）。虽然类分组符号描述了类或组件集合中存在的基本继承和包容关系，但整个类关系图定义了这些类与组件间存在的整体关系。

类可以用于关联函数的代码与数据。执行这个函数的类称做接口类（interface class），它用于为代码和数据提供新接口，或者更新旧的接口。接口类可以用于为非面向对象编程工作（如过程库、操作系统 API，或者数据库管理系统）提供面向对象接口。类充当现有过程和数据的封装器。接口类将这些过程和数据封装起来，并提供仅能被它们访问的成员函数。接口类也可用于修改或改进另一个类或类集合的接口。当接口类用作此目的时，它称做适配类（adaptor class）。适配类使其他类更易于使用，功能更强，更安全，或者从语义上更准确。适配类调整或优化另一个类的接口，使它更有用或效率更高。

存在几种接口类，每种接口类均服务于不同的函数。这种接口可以通过继承或包容创建。当代码和数据或类被继承时，就使用继承符号和关系运算符。当代码、数据或类在分组符号中表示时，它要么表示为类的一个继承组件，要么表示为类的一个包容组件。它带有阴影及内指的箭头，以区别于其他继承或包容组件。这个类创建一个代码与数据或者类的接口。内指箭头表示，接口类将对该组件的函数或成员函数作出函数调用。

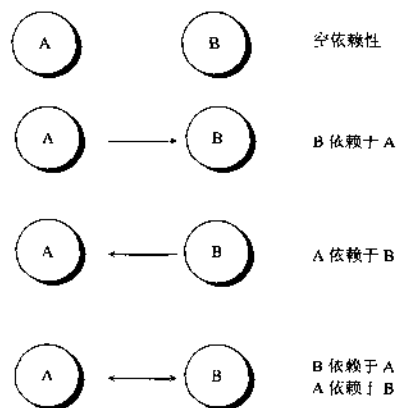
友元函数是一个可以访问类隐藏成员的非成员函数。友元函数是类的一部分，它在类的头中声明。友元函数与类没有包容或继承关系。它不是类的实际组件。友元表达为类的外界对象。双向箭头表示友元与类的通信，以及类与友元函数的通信。

线程依赖性图与矩阵

线程依赖性图（Thread Dependency Graph）用于注释线程或进程集合间的关系。依赖性图可以表达应用中哪些线程具有依赖性，而不需要搜索整个源代码来确定这些关系。这些信息可用于理解、维护和测试多线程程序。图 B-1 中显示了两个线程或进程间的依赖关系。依赖性图可以显示单向或双向依赖性。单向依赖性意味着仅在一个方向存在依赖性。双向依赖性意味着在两个方向都存在依赖性。空依赖性意味着在线程或进程间不存在依赖性。依赖性图可以用于线程和进程数量较少的场合。图 B-1 显示了存在的依赖性，但没有显示依赖性的类型。

依赖性矩阵可用于诠释较大线程或进程集合间的关系。图 B-1 也使用依赖性矩阵显示了 3 个线程或进程间的依赖性关系。线程或进程行依赖于线程和进程列。矩阵可以显示单向依赖性，也可以显示双向依赖性，以及存在的依赖性类型。例如，在图 B-1 中，线程 A 依赖于线程 B 和线程 C。I 意味着信息依赖性，C 意味着通信依赖性，S 意味着同步依赖性。线程 A 对线程 B 和线程 C 存在信息依赖性。线程 C 也对线程 A 存在信息依赖性。所以，线程 A 和线程 C 具有双向信息依赖性。另一方面，线程 A 和线程 B 有单向依赖性。线程 A 对线程 B 存在信息依赖性，但线程 B 对线程 A 不存在通信依赖性。对线程 B 和线程 C 也是这样。线程 B 对线程 C 存在同步依赖性，

但线程 C 对线程 B 不存在通信依赖性。



A 与 B 间的依赖性矩阵

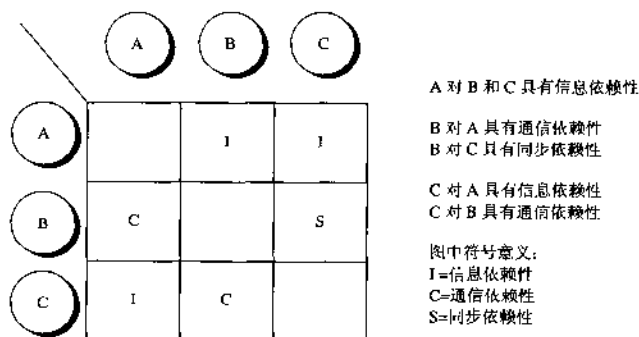


图 B-1 图中描绘了两个线程或进程间的依赖性关系，在矩阵中描绘了三个进程或线程间的依赖性关系。依赖性图和矩阵显示了单向和双向依赖性。空依赖性意味着线程或进程间不存在依赖性。矩阵可用于显示存在的关系以及庞大线程或进程集合间的关系类型

附录 C POSIX 线程管理函数

线程属性

属 性	描 述
<i>contentionscope</i>	决定线程对资源竞争级别：进程作用域或系统作用域。进程中的线程与其他线程竞争资源。线程也可以与系统中全局性分配的其他进程中的线程竞争资源。这个属性可以通过线程库在局部作用域内规划，也可以通过操作系统在全局作用域内规划。
<i>detachstate</i>	允许其他线程等待特定线程的终止。
<i>stackaddr</i>	设置指向线程堆栈的指针。
<i>stacksize</i>	设置线程的堆栈的大小。
<i>priority</i>	设置规划类内线程的优先权。
<i>policy</i>	设置规划策略（FIFO，轮询及其他）
<i>inheritsched</i>	决定规划参数是否被继承或定义。

线程属性函数

int pthread_attr_int(pthread_attr_t *attr)

参数

***attr** 指向线程属性对象的指针。

返回值

如果成功，返回 0。不成功，返回一个错误号。

描述

初始化线程属性对象。属性对象可用于创建其他新线程或修改它。它被传递给 `pthread_create` 函数。

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)

参数

***attr** 指向线程属性对象的指针。

detachstate 判断线程是否以分离状态创建。这个属性可以有以下几种值：

PTHREAD_CREATE_DETACHED 创建一个新分离线程。当销毁一个分离线程时，它不会脱离跟踪。释放线程 ID 和其他所有资源以备再次使用。

PTHREAD_CREATE_JOINABLE 创建一个新非分离线程。线程 ID 和用户自定义堆栈直到调用函数 `pthread_join()` 时才被释放；在终止时，为了释放与该线程关联的所有资源，必须调用 `pthread_join()`。

返回值

如果成功，返回 0。不成功，返回一个错误号。

描述

设置属性对象中的 `detachstate` 属性。

int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate)

参数

***attr** 指向线程的属性对象的指针。

detachstate 判断线程是否以分离状态创建。这个属性可以有以下几种值：

PTHREAD_CREATE_DETACHED 创建为分离线程。

PTHREAD_CREATE_JOINABLE 创建为非分离线程。

返回值

如果成功，返回线程的 `detachstate` 并返回 0。如果不成功，返回一个错误号。

描述

获取属性对象的 `detachstate` 属性。

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched)

参数

***attr** 指向线程属性对象的指针。

inheritsched 判断线程的规划策略。这个属性可以有以下几种值：

PTHREAD_INHERIT_SCHED 设置新线程的规划策略和参数与正创建线程相同。忽略线程创建属性结构中的规划策略和规划参数属性。

PTHREAD_EXPLICIT_SCHED 根据线程创建属性结构中的规划策略和规划参数属性来设置新线程的规划策略和规划参数。

返回值

如果成功，返回 0。如果不成功，返回一个错误号。

描述

设置线程的规划继承。线程可以继承创建者线程的规划策略和参数，也可以从线程创建函数中指定的线程属性对象获取规划策略。

int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inheritsched)

参数

***attr** 指向线程属性对象的指针。

inheritsched 线程的规划策略。它可以为以下几种值：

PTHREAD_INHERIT_SCHED 将线程的规划策略和参数设置为与创建线程相同。

PTHREAD_EXPLICIT_SCHED 根据线程创建属性结构中的规划策略和规划参数属性来设置线程的规划策略和参数。

返回值

如果成功，返回线程的规划策略并返回 0。如果不成功，返回一个错误号。

描述

获取线程的规划继承。

int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param)

参数

***attr** 指向线程属性对象的指针。

***param** param 结构。此结构中唯一使用的成员是 sched_priority。缺省时，其值为 NULL，它意味着新线程继承了父线程的优先权。

返回值

如果成功, 返回 0。如果不成功, 返回一个错误号。

描述

设置创建线程的优先权。参数的值取决于规划策略:

SCHED_FIFO
SCHED_RR
SCHED_OTHER

int pthread_attr_getschedparam(pthread_attr_t *attr, const struct sched_param *param)

参数

*attr 指向线程属性对象的指针。

*param param 结构。这一结构中唯一使用的成员是 sched_priority。

返回值

如果成功, 则返回规划策略参数并返回 0。如果不成功, 则返回一个错误号。

描述

获取线程的优先权。它的值取决于规划策略:

SCHED_FIFO
SCHED_RR
SCHED_OTHER

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)

参数

*attr 指向线程属性对象的指针。

*policy 策略属性的指针。它的值有以下几种:

SCHED_FIFO
SCHED_RR
SCHED_OTHER (缺省值)

返回值

如果成功, 则返回 0。如果不成功, 则返回一个错误号。

描述

设置线程的规划策略属性。

int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)

参数

***attr** 指向线程属性对象的指针。

***policy** 策略属性指针。它的值有以下几种：

SCHED_FIFO

SCHED_RR

SCHED_OTHER (缺省值)

返回值

如果成功，则返回线程的策略属性并返回 0。如果不成功，则返回一个错误号。

描述

获取线程的规划策略属性设置。

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope)

参数

***attr** 指向线程属性对象的指针。

contentionscope 线程的竞争作用域。竞争作用域决定线程竞争资源的级别。它的值有以下几种：

PTHREAD_SCOPE_SYSTEM 线程将与它的进程的线程以及其他进程的线程竞争资源。这是缺省值。

PTHREAD_SCOPE_PROCESS 线程与它的进程的其他线程竞争资源。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

设置线程的竞争作用域。线程可以在系统作用域竞争资源，此时，操作系统比较系统范围内所有进程的所有可运行线程的优先权，选择运用处理器的线程。线程可以在进程作用域竞争资源，此时，操作系统比较进程范围内所有可运行线程的优先权，选择运用处理器的线程。只让最高优先权的线程使用处理器。

int pthread_attr_getscope(pthread_attr_t *attr, int *contentionscope)

参数

***attr** 指向线程属性对象的指针。

contentionscope 线程的竞争作用域。竞争作用域决定线程竞争资源的级别。它具有以下几种值：

PTHREAD_SCOPE_SYSTEM	线程与该进程的线程以及其他进程的线程竞争资源。
PTHREAD_SCOPE_PROCESS	线程与该进程的其他线程竞争资源。

返回值

如果成功，返回线程的竞争作用域，并返回 0。如果不成功，则返回一个错误号。

描述

获取线程的竞争作用域。

int pthread_attr_setstackaddr(const pthread_attr_t *attr, void *stackaddr)

参数

*attr 指向线程属性对象的指针。

*stackaddr 线程堆栈地址的指针。缺省值为 NULL。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

调整线程的堆栈地址。

int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr)

参数

*attr 指向线程属性对象的指针。

**stackaddr 指向线程堆栈地址的指针的指针。

返回值

如果成功，返回则堆栈地址和 0。如果不成功，则返回一个错误号。

描述

获得线程的堆栈地址。

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)

参数

*attr 指向线程属性对象的指针。

stacksize 线程的堆栈大小。缺省参数为 NULL，缺省大小为 1MB。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

设置线程的堆栈大小。这个大小必须大于或等于 PTHREAD_STACK_MIN。

```
int pthread_attr_getstacksize(const pthread_attr_t *attr,  
                             size_t *stacksize)
```

参数

***attr** 指向线程属性对象的指针。

***stacksize** 线程堆栈大小的指针。

返回值

如果成功，返回线程的堆栈大小并返回 0。如果不成功，则返回一个错误号。

描述

获取线程的堆栈大小。

```
int pthread_attr_destroy(pthread_attr_t *attr)
```

参数

***attr** 指向线程属性对象的指针。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

销毁线程属性对象。直到重新初始化后才能使用它。

线程创建函数

```
int pthread_create(pthread_t *new_thread_ID, const pthread_attr_t *attr, void *  
(*start_func)(void *), void *arg)
```

参数

***new_thread_ID** 新线程标识的指针。

***attr** 指向线程属性对象的指针。

***start_func** 用户自定义函数的地址，它是新线程的执行路线。

***arg** 线程函数的参数。

返回值

如果成功，则返回 0。它也返回线程的 ID。如果函数不成功，则返回一个非零值。

描述

在某进程内创建一个新线程，将其属性保存在属性对象中。如果在创建线程前，参数 `attr` 指定的任何属性在属性对象中被更改了，新创建的线程将采用这些更新值。如果更改发生在创建线程之后，则它们不会影响该线程。线程属性的缺省值如表 B-1。

表 B-1 线程属性的缺省值表

属 性	缺 省 值	意 义
<code>contentionscope</code>	<code>PTHREAD_SCOPE_PROCESS</code>	线程在进程级别竞争资源
<code>detachstate</code>	<code>PTHREAD_CREATE_JOINABLE</code>	线程被其他线程合并
<code>stackaddr</code>	<code>NULL</code>	堆栈由系统分配
<code>Stacksize</code>	<code>NULL</code>	堆栈大小为 1MB
<code>priority</code>	—	线程具有父亲或调用线程的优先权
<code>policy</code>	<code>SCHED_OTHER</code>	由系统决定
<code>inheritsched</code>	<code>PTHREAD_EXPLICIT_SCHED</code>	规划策略和参数不被继承。它们将由线程的属性对象定义

线程终止/取消函数

终止函数

```
void pthread_exit(void *status)
```

参数

`*status` 线程的退出状态。

返回值

无返回值。

描述

终止调用线程。如果调用线程不处理分离状态，则获得参数指定的线程 ID 和退出状态。与终止线程成功合并的任何线程都可以访问该线程的状态。

```
int pthread_join(pthread_t target_thread, void **status)
```

参数

target_thread 线程的句柄
****status** 指向终止线程返回值的指针的指针。

返回值

如果成功，返回一个非空的状态值，并返回 0。如果不成功，则返回一个错误号。

描述

挂起调用线程的过程，直到指定线程完成为止。指定线程必须与调用线程同属于一个进程。它不能为分离或守护线程。

```
int pthread_detach(pthread_t threadID)
```

参数

threaded 被分离线程的句柄

返回值

如果成功，则返回 0。如果不成功，则返回一个表征该错误的非零值。

描述

标记要删除的指定线程的内部数据结构。将线程的分离状态属性重置为 PTHREAD_CREATE_DETACHED。当分离线程终止时，系统将回收线程对象所使用的存储空间。

取消函数

```
int pthread_cancel(pthread_t target_thread)
```

参数

target_thread 被取消线程的句柄

返回值

如果取消成功，则返回 0。如果不成功，则返回一个错误号。

描述

让一个线程取消进程中的另一个线程。不立即取消线程。它继续执行，直到到达取消点（调

用某程序)为止。线程检查取消是否发生。如果发生了,则取消线程。

下面是所需要的 POSIX 取消点程序:

aio_suspend()	read()
close()	sem_wait()
creat()	sigsuspend()
fcntl(, F_SETLK,)	sigtimedwait()
fsync()	sigwait()
mq_receive()	sigwaitinfo()
mq_send()	sleep()
msync()	system()
nanosleep()	tcdrain()
open()	wait()
pause()	waitpid()
pthread_cond_timedwait()	write()
pthread_cond_wait()	
pthread_join()	
pthread_testcancel()	

int pthread_setcancelstate(int state, int *oldstate)

参数

state 将线程的取消状态设置为该状态。
oldstate 保存前一个取消状态。

返回值

如果取消成功,则返回 0。如果不成功,则返回一个错误号。

描述

设置线程的可取消状态。如果取消状态为禁止,则线程不会对待决的取消请求作出反应。这些状态为:

PTHREAD_CANCEL_ENABLE 对取消请求作出反应。

PTHREAD_CANCEL_ASYNCHRONOUS 如果取消状态为允许,待决或取消点在任何时候都能起作用。

PTHREAD_CANCEL_DISABLE 取消请求保持待决。

PTHREAD_CANCEL_DEFERRED 取消请求在到达某个取消点前保持待决。

int pthread_setcanceltype(int type, int *oldtype)

参数

type 将调用线程的取消类型设置为该类型。
oldtype 保存前一个取消类型。

返回值

如果取消成功，则返回 0。如果不成功，则返回一个错误号。

描述

设置线程的取消类型。它允许线程在取消点接收取消命令。当取消状态为禁止时，取消类型无意义。取消类型有以下几种：

PTHREAD_CANCEL_ASYNCHRONOUS

`pthread_cancel()` 调用导致即时线程取消。

PTHREAD_CANCEL_DEFERRED 直到线程到达取消点后，才会发生线程的取消。

`pthread_cancel()`调用将导致取消点发生线程取消。

void pthread_testcancel()

参数

无参数。

返回值

无返回值。

描述

强迫取消测试。它请求将所有待决取消请求发送给调用线程。

void pthread_cleanup_pop(int execute)

参数

execute 如果该值为非零，则执行取消清理堆栈顶端的清理程序。

返回值

无返回值。

描述

删除调用线程的取消清理堆栈顶端的清理处理程序。如果参数为非零，则执行该程序。

void pthread_cleanup_push(void (*handler)(void *), void *arg)

参数

***handler** 指向清理程序的描述符。

***arg** 指向清理程序参数的指针。

返回值

无返回值。

描述

将清理程序推入线程的清理堆栈顶端。当调用此程序时，将参数传递给函数。

线程管理函数

规划/优先权函数

int pthread_setschedparam(pthread_t target_thread, int policy, const struct sched_param *param)

参数

target_thread 线程句柄

policy 规划策略

*param param 结构。这个结构中唯一使用的成员是 sched_priority。

返回值

如果函数成功，则返回 0。如果不成功，则返回一个错误号。

描述

设置目标线程的规划策略和相关优先权。设置策略以及该策略提供的相关优先权。如果函数失败，不更改指定线程中的规划参数。

pthread_getschedparam(pthread_t target_thread, int *policy, struct sched_param *param)

参数

target_thread 线程句柄。

*policy 规划策略指针。

*param param 结构。这个结构中唯一使用的成员是 sched_priority。

返回值

如果函数成功，则返回 0。如果函数不成功，则返回一个错误号。

描述

获得指定线程的规划策略和优先权参数。策略保存在策略参数中，优先权保存在 param 结构的 sched_priority 成员中。

线程特定信息函数

int pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *value))

参数

*keyp 键指针。
 *destructor 析构函数的指针。
 *value 与键绑定的值。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

创建一个键。键定位特定于进程中每个线程的数据。它对于进程的所有线程是全局性的。每个线程一旦创建一个键后，它就将某个值绑定到该键。对于每个绑定的线程，这个键保持指定的值。析构函数可以与每个键关联。当这个线程存在时，如果该键有一个非 NULL 析构函数，而且线程有一个与键关联的非 NULL 值，则通过该值调用这个函数。

int pthread_setspecific(pthread_key_t key, const void *value)

参数

key 键。
 *value 绑定于键值的指针。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

创建键后，绑定指定新值到该键。参数值为指向调用线程保留的动态分配内存块的指针。

void *pthread_getspecific(pthread_key_t key)

参数

key 键。

返回值

返回调用线程指定键所绑定的当前值。如果这个键没有绑定的值，则返回 NULL 值。

描述

获得调用线程的参数中指定键关联的特定线程值。

int pthread_key_delete(pthread_key_t key)

参数

key 键。

返回值

如果函数成功，则返回 0。如果函数不成功，则返回一个错误号。

描述

删除特定线程键。

线程 ID 函数

pthread_t pthread_self(void)

参数

无参数。

返回值

返回线程的标识符。

描述

返回调用线程的线程 ID。

int pthread_equal(pthread_t t1, pthread_t t2)

参数

t1 线程句柄。

t2 线程句柄。

返回值

如果线程标识符相等，则返回一个非零值。否则，返回 0。

描述

比较指定线程的线程标识符。

线程信号函数

int pthread_kill(pthread_t thread, int sig)

参数

thread 线程句柄。

sig 信号。

返回值

如果函数成功，则返回 0。如果函数不成功，则返回一个错误号。如果发生错误，不发送信号。

描述

将信号发送到指定线程。线程必须与调用线程同属于一个进程。如果信号为 0，则验证线程是否存在。实际上无信号发送。

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset)
```

参数

how 指定更改 **set** 的方法。
***set** 指向一个可以修改当前阻塞设置的 **set** 信号。
***oset** 指向保存前一个信号掩码的位置。

返回值

如果成功，则返回 0。如果不成功，则返回一个非零值。

描述

更改和（或）检查调用线程的信号掩码。新线程将继承调用线程的信号掩码以及它的优先权。如果存在待决信号，则不继承它们。参数的值有以下几种：

SIG_SETMASK 对应于新信号掩码的设置。当前信号掩码由 **set** 设置。
SIG_BLOCK 与被阻塞的一套信号对应的 **set** 参数。它们被添加到当前信号掩码上。
SIG_UNBLOCK 与未被阻塞的一套信号对应的 **set** 参数。从当前信号掩码中将它们删除。

互斥量函数

初始化/创建函数

```
int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *attr)
```

参数

***mp** 被初始互斥量的指针。
***attr** 包含用于互斥量属性的数据结构体的指针。

返回值

如果函数成功，则返回 0。如果不成功，则返回一个错误号。

描述

初始化一个互斥量。互斥量的状态被初始化并被取消阻塞。

析构函数

```
int pthread_mutex_destroy(pthread_mutex_t *mp)
```

参数

*mp 被销毁互斥量的指针。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

销毁指定互斥量。互斥量必须被初始化。如果该互斥量正被一个阻塞线程等待，则返回一个错误。

互斥量属性

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr)
```

参数

*attr 包含用于互斥量属性的数据结构的指针。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

用缺省值初始化互斥量属性对象。

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)
```

参数

*attr 包含用于互斥量属性的数据结构的指针。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

销毁互斥量属性对象。该对象变成未初始化。

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int process-shared)
```

参数

***attr** 包含用于互斥量属性的数据结构指针。

process-shared 允许同步变量被两个或更多进程的线程联合共享的线程属性。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误号。

描述

在参数 **attr** 指定的属性对象中设置共享进程的属性。

```
int pthread_mutexattr_getpshared(pthread_mutexattr_t *attr, int process-shared)
```

参数

***attr** 包含用于互斥量属性的数据结构指针。

process shared 允许同步变量被两个或更多进程中线程联合共享的线程属性。

返回值

如果成功，则返回 0 并将值保存在共享进程参数中；如果不成功，则返回一个错误号。

描述

获取参数 **attr** 指定的属性对象中的共享进程属性。

```
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling)
```

参数

***attr** 互斥量属性对象的指针。

prioceiling 包含优先权极限的互斥量属性。

返回值

如果函数成功，则返回 0。如果不成功，则返回一个错误号。

描述

设置互斥量属性对象的 **prioceiling** 属性。

锁定函数

```
int pthread_mutex_lock(pthread_mutex_t *mp)
```

参数

*mp 被锁定的互斥量指针。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误值。

描述

试图锁定互斥量。它将导致另一个试图锁定同一个互斥量的线程阻塞，直到互斥量的占有者取消它的锁定为止。如果互斥量没有正确初始化，则返回一个错误。

```
int pthread_mutex_trylock(pthread_mutex_t *mp)
```

参数

*mp 互斥量的指针。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误值。

描述

试图锁定互斥量。如果互斥量已被锁定，调用线程将不等待互斥量的取消锁定而返回。

```
int pthread_mutex_unlock(pthread_mutex_t *mp)
```

参数

*mp 取消锁定互斥量的指针。

返回值

如果成功，则返回 0。如果不成功，则返回一个错误值。

描述

试图解锁互斥量。只有互斥量的占有者才能取消锁定互斥量。如果调用线程不是互斥量的占有者，它可能返回一个错误。如果调用线程为占有者，互斥量将被取消锁定。另一个线程可以锁定该互斥量。规划优先权将决定哪一个等待该互斥量的阻塞线程被恢复，并允许锁定这个互斥量。这个锁定过程可能成功，也可能不成功。

条件变量函数

属性函数

```
int pthread_condattr_init(pthread_condattr_t *attr)
```

参数

***attr** 条件变量属性对象的指针。

返回值

如果函数成功，则返回 0。如果发生错误，则返回一个非零值。

描述

初始化由参数指定的条件变量属性对象。这个属性由缺省值来初始化。一旦通过这个属性对象初始化了其他变量对象，影响这个属性的任何函数不会影响前面初始化的条件变量。

int pthread_condattr_setpshared(pthread_condattr_t *attr, int process-shared)

参数

***attr** 条件变量属性对象的指针。

process-shared 允许同步变量被两个或多个进程中线程联合共享的线程属性。

返回值

如果函数成功，则返回 0。如果发生错误，则返回一个非零值。

描述

在参数 **attr** 指定的初始化条件变量属性对象中设置 **process-shared** 属性。

PTHREAD_PROCESS_PRIVATE 只允许在初始化条件变量的同一个进程中创建的线程作用于条件变量。

PTHREAD_PROCESS_SHARED 允许任何可以访问分配给条件变量内存的线程作用于条件变量。

int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *process-shared)

参数

***attr** 条件变量属性对象的指针。

***process-shared** 允许同步变量被两个或更多进程的线程联合共享的线程属性指针。

返回值

如果函数成功，则返回 0 并将条件变量属性对象指定的 **process-shared** 属性值保存在参数 **attr** 中。这个值保存在参数 **process-shared** 中。如果发生错误，则返回一个非零值。

描述

从参数 **attr** 指定的条件变量属性对象中获取属性 **process-shared** 的值。

int pthread_condattr_destroy(pthread_condattr_t *attr)

参数

*attr 条件变量属性对象的指针。

返回值

如果函数成功，则返回 0。如果发生错误，则返回一个非零值。

描述

销毁参数指定的条件变量对象。这个对象变成未初始化。

初始化函数

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
```

参数

*cond 被初始化条件变量的指针。

*attr 条件变量属性对象的指针。

返回值

如果函数成功，则返回 0。如果发生错误，则返回一个非零值。

描述

通过这个参数所引用的属性对象中指定的属性来初始化条件变量。如果 attr 为 NULL，则使用缺省条件变量属性对象。初始化后，条件变量状态为已初始化。

等待函数

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

参数

*cond 条件变量指针。

*mutex 互斥量指针。

返回值

如果函数成功，则返回 0。如果发生错误，则返回一个非零值。

描述

阻塞于释放特定互斥量的条件变量上。它导致调用线程阻塞于该条件变量。当条件变量被信号通知时，互斥量被释放，而且被调用线程占有和锁定。

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct
timespec *abstime)
```

参数

*cond 条件变量指针。
*mutex 互斥量指针。
*abstime 线程必须等待条件变量的时间量指针。

返回值

如果函数成功，则返回 0。如果发生错误，则返回一个非零值。

描述

阻塞于释放指定互斥量的条件变量上。它导致调用线程阻塞于该条件变量。当条件变量被信号通知时，互斥量被释放，而且被调用线程占有和锁定。线程将等待条件变量一定时间。如果系统时间等于或超过这个时间，就返回一个错误。

信号/发送函数

```
int pthread_cond_signal(pthread_cond_t *cond)
```

参数

*cond 条件变量指针。

返回值

如果函数成功，则返回 0。如果发生错误，则返回一个非零值。

描述

如果任何线程阻塞于这个条件变量，这个函数至少取消阻塞一个线程。规划优先权决定哪一个线程被取消阻塞。调用线程不必为占有者。如果不存在阻塞于条件变量的线程，则它无影响。

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

参数

*cond 条件变量的指针。

返回值

如果函数成功，则返回 0。如果发生错误，则返回一个非零值。

描述

如果有线程阻塞于条件变量，这个函数取消阻塞所有线程。调用线程不必为占有者。如果不存在阻塞于条件变量的线程，则它无影响。

析构函数

int pthread_cond_destroy(pthread_cond_t *cond)

参数

*cond 条件变量的指针。

返回值

如果函数成功，则返回 0。如果发生错误，则返回一个非零值。

描述

销毁指定条件变量。不释放空间。

管道函数

管道创建函数

UNIX

int pipe(int fd[])

参数

fb[] 文件描述符

返回值

返回 2 个文件描述符：fd[0]保存与读取管道末尾相关的文件描述符；fd[1]保存与写管道末尾相关的文件描述符。如果不能分配管道，这个管道函数将返回-1。如果成功，函数返回 0。

描述

创建一个未命名管道并返回两个文件描述符：fd[0]保存与读取管道末尾相关的文件描述符；fd[1]保存与写管道末尾相关的文件描述符。如果进程读取管道，则必须关闭写末尾。如果进程读取一个空管道，而且写末尾仍然打开，则进程会一直休眠到某些输入可用为止。如果进程试图从管道读取比实际存在的字节更多的字节，则返回当前所有内容，而且 read()函数将返回实际读取的字节数。

如果进程写入管道，读末尾需要关闭。如果没有关闭它，写操作就会失败，而且发送写入者一个 SIGPIPE 信号。缺省时，它将终止接收者。如果进程写入的字节比管道可以容纳的字节少，可以确保 write() 的原子性。写入者进程将完成系统调用，而不会被其他进程抢占。如果进程写入管道的字节比管道可以容纳的字节多，就不能得到这种原子性的保证。

附录 D Win32 线程管理函数

线程属性函数

BOOL GetThreadContext(**HANDLE** *hThread*, **LPCONTEXT** *lpContext*)

参数

hThread 标识上下文被读取的线程打开句柄。句柄必须具有访问线程的 **THREAD_GET_CONTEXT** 权限。

lpContext 指向 **CONTEXT** 结构的地址。

返回值

如果成功，则返回 **TRUE**。如果不成功，则返回 **FALSE**。

描述

获取指定线程的上下文。这个函数用于读取指定线程的上下文。函数允许基于 **CONTEXT** 结构的 **ContextFlags** 成员选择性地读取上下文。**CONTEXT** 结构的 **ContextFlags** 成员的值指定读取线程的哪一部分上下文。

线程创建

HANDLE CreateThread(**LPSECURITY_ATTRIBUTES** *lpSa*, **DWORD** *cbStack*,
LPTHREAD_START_ROUTINE *lpStartAddr*, **LPOVOID** *lpvThreadParm*, **DWORD** *fdwCreate*,
LPDWORD *lpIDThread*)

参数

lpSa

指向 SECURITY_ATTRIBUTES，它指定线程的安全性属性。

cbStack 指定堆栈的大小（以字节为单位）。

lpStartAddr 指向新线程即将执行的函数。

lpvThreadParm 指定传递给函数的参数。

fdwCreate 指定控制新线程创建的额外标志。

lpIDThread 指向线程标识符。

返回值

如果成功，则返回新线程的句柄。如果不成功，返回值为 NULL。

描述

创建一个线程。这个函数创建一个管理和标识线程的线程核心对象。它也为线程分配上下文结构、准备堆栈并初始化堆栈指针寄存器及指令指针寄存器。如果参数 *lpSa* 为 NULL，则通过缺省安全性描述符创建线程。如果堆栈参数为 0，则线程的新堆栈大小与当前进程所用堆栈大小相同。可以在挂起状态创建线程。如果指定了 CREATE_SUSPENDED 标志，则线程创建于挂起状态，直到调用恢复线程函数后才能运行。

**HANDLE CreateRemoteThread(HANDLE *hProcess*, LPSECURITY_ATTRIBUTES *lpSa*,
DWORD *cbStack*, LPTHREAD_START_ROUTINE *lpStartAddr*, LPVOID *lpvThreadParm*,
DWORD *fdwCreate*, LPDWORD *lpIDThread*)**

参数

hProcess 创建线程所在进程的句柄。

lpSa 指向 SECURITY_ATTRIBUTES 结构，它指定线程的安全性属性。

cbStack 指定堆栈的大小（以字节为单位）。

lpStartAddr 指向新线程即将执行的函数。

lpvThreadParm 指定传递给函数的参数。

fdwCreate 指定控制新线程创建的额外标志。

lpIDThread 指向线程标识符。

返回值

如果成功，则返回新线程的句柄。如果不成功，返回值为 NULL。

描述

创建一个在另一个进程的地址空间中运行的线程。如果参数 *lpSa* 为 NULL，则通过缺省安全性描述符创建线程。如果堆栈参数为 0，则线程的新堆栈大小与当前进程所用堆栈大小相同。可

以在挂起状态创建线程。如果指定了 `CREATE_SUSPENDED` 标志，则线程创建于挂起状态，直到调用 `ResumeThread()` 后才能运行。

线程终止函数

VOID ExitThread(DWORD dwExitCode)

参数

dwExitCode 退出码

返回值

无返回值。

描述

终止线程并设置线程的退出码为参数。这个函数导致当前线程堆栈的释放，而且线程终止。如果调用这个函数时，该线程为进程中的最后一个线程，进程也将终止。线程对象的状态变成信号通知状态。当线程对象被信号通知时，其他等待着该线程终止的线程将被释放。线程的终止状态从 `STILL_ACTIVE` 变为 `dwExitCode` 参数的值。终止一个线程不需要从操作系统删除线程对象。在关闭线程的最后一个句柄时，线程对象才被删除。

BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode)

参数

hThread 被终止的线程句柄。

dwExitCode 退出码。

返回值

如果成功，则返回 `TRUE`。如果不成功，则返回 `FALSE`。

描述

终止指定线程并设置退出码为 `dwExitCode`。在线程不再作出反应时必须使用它。终止线程不需要从系统删除线程对象。在关闭最后一个线程句柄时，线程对象才被删除。

BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpdwExitCode)

参数

hThread 线程句柄。

lpdwExitCode 指向特定线程的退出码。

返回值

如果成功,则返回 TRUE。如果调用 GetExitCodeThread 后,线程没有终止,函数将返回 DWORD STILL_ACTIVE 标识符。

描述

获取指定线程的退出码。

线程管理函数

优先权函数

BOOL SetThreadPriority(HANDLE hThread, int nPriority)

参数

hThread 更改优先类的线程句柄。

nPriority 其值有以下之一:

THREAD_PRIORITY_LOWEST 线程的优先权必须比进程的优先类小 2。

THREAD_PRIORITY_BELOW_NORMAL 线程的优先权必须比进程的优先类小 1。

THREAD_PRIORITY_NORMAL 线程的优先权必须与进程的优先类相等。

THREAD_PRIORITY_ABOVE_NORMAL 线程的优先权必须比进程的优先类大 1。

THREAD_PRIORITY_HIGHEST 线程的优先权必须比进程的优先类大 2。

THREAD_PRIORITY_IDLE 不管进程的优先类是否为空闲、普通,还是高,将线程的优先级设为 1。如果进程的优先类为实时, **THREAD_PRIORITY_IDLE** 设置线程的优先级设为 16。

THREAD_PRIORITY_TIME_CRITICAL 不管进程的优先类是否为空闲、普通,还是高,将线程的优先级设为 15。如果进程的优先权为实时,设置线程的优先级设为 31。

返回值

如果成功,则返回 TRUE。如果不成功,则返回 FALSE。

描述

更改单个进程内线程的相对优先权。每个线程有一个基优先级,它由线程的优先权值与进程的优先类决定。可执行线程的基优先级决定哪一具线程使用 CPU。在每个优先级,按轮巡方式规划所有线程,仅当较高级别没有可执行线程时,才会进行较低级别的线程规划。这个函数允许相对于进程的优先类来设置线程的基优先级。

int GetThreadPriority(HANDLE hThread)

参数

hThread 线程的句柄

返回值

如果成功，返回线程的优先标识符。如果发生错误，返回 `THREAD_PRIORITY_ERROR_RETURN`。

描述

返回指定线程的优先权。

挂起/恢复函数

DWORD SuspendThread(HANDLE hThread)

参数

hThread 被挂起线程的句柄。这个句柄必须具有 `THREAD_SUSPEND_RESUME` 访问权限。

返回值

如果成功，返回线程的前一个挂起计数。如果不成功，返回值 `0xFFFFFFFF`。

描述

挂起指定线程。如果函数成功，指定线程的执行被挂起，并增加线程的挂起计数。线程挂起导致线程停止执行用户模式（应用）代码。每个线程有一个挂起计数。它有一个最大值 `MAXIMUM_SUSPEND_COUNT`。如果挂起计数大于 0，则挂起这个线程。如果不是这样，则不挂起线程，它有资格执行。如果试图增加挂起计数超过其最大值，则发生错误，而不会增加这个计数。

DWORD ResumeThread(HANDLE hThread)

参数

hThread 被恢复线程的句柄。

返回值

如果成功，返回线程的前一个挂起计数。如果不成功，返回值 `0xFFFFFFFF`。

描述

恢复挂起线程。挂起线程可能是以挂起状态创建的线程，也可能是从创建后就被挂起的线程。

线程特定信息函数

BOOL GetThreadTimes(HANDLE hThread, LPFILETIME lpCreationTime, LPFILETIME lpExitTime, LPFILETIME lpKernelTime, LPFILETIME lpUserTime)

参数

- hThread** 一个指定所求定时信息的线程的打开句柄。
lpCreationTime 指向接收线程创建时间的 FILETIME 结构。
lpExitTime 指向接收线程退出时间的 FILETIME 结构。
lpKernelTime 指向接收线程在核心模式中已执行时间的 FILETIME 结构。
lpUserTime 指向接收线程在用户模式中已执行时间的 FILETIME 结构。

返回值

如果成功，则返回 TRUE。如果不成功，则返回 FALSE。

描述

获得指定线程的定时信息。所有的时间使用 FILETIME 数据结构来表达。线程创建和退出时间为英格兰的格林尼治标准时间 1/1/1601 午夜到计算点所经过的时间段。

DWORD SetThreadAffinityMask (HANDLE hThread, DWORD dwThreadAffinityMask)

参数

- hThread** 由函数设置亲和掩码的线程句柄。
dwThreadAffinityMask 为线程指定亲和掩码。

返回值

如果成功返回非零值。在 Windows NT 下，返回值为线程的前一个亲和掩码。如果不成功，则返回 0。

描述

为指定线程设置处理器亲和掩码。线程的亲和掩码是一个位矢量，其中的每个位表示允许线程运行的处理器。线程仅允许在它的进程允许运行的处理器上运行。

线程 ID 函数

DWORD GetCurrentThreadID(VOID)

参数

无参数。

返回值

返回调用线程的线程 ID。

描述

获得调用线程的线程 ID。直到线程终止，线程标识符在整个系统中唯一地标识该线程。

DWORD GetCurrentThread(VOID)

参数

无参数。

返回值

返回一个伪句柄，它仅当用于当前线程上下文中进才有意义。

描述

获取当前线程的伪句柄。这个函数不能被线程用于创建一个可被其他线程用于引用第一个线程的句柄。这个句柄引用正使用它的线程。当不再需要伪句柄时，需要关闭它。

互斥量函数

创建函数

HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpsa, BOOL fInitialOwner, LPCSTR lpszMutexName)

参数

lpsa SECURITY_ATTRIBUTES 结构的指针。

fInitialOwner 如果为 TRUE，调用线程请求被创建互斥量对象的立即占有权。如果为 FALSE，互斥量不被初始占有。

lpszMutexName 指向一个指定互斥量对象名字的字符串。

返回值

返回互斥量对象的一个句柄。如果命名互斥量对象在函数调用前存在，则 GetLastError 函数返回 ERROR_ALREADY_EXISTS。否则，GetLastError 返回 0。

描述

创建一个命名或未命名互斥量对象。如果命名参数为 NULL，则不使用名字来创建互斥量。

调用进程的任何线程都可以在调用其中一个等待函数的过程中指定互斥量对象句柄。当互斥量对象被占有时，互斥量对象的状态为信号通知。当互斥量的状态为信号通知时，授予一个等待线程占有权，互斥量的状态改为非信号通知，等待函数返回。某一时刻仅有一个线程可以占有互斥量。占有互斥量的线程使用 `ReleaseMutex()` 来释放它的占有权。两个或多个进程可以调用这个函数来创建相同的命名互斥量。第一个进程创建互斥量，另外的进程打开现有互斥量的句柄。

这允许多个进程得到同一个互斥量的句柄，同时减轻了用户确保必须首先启动创建进程的负担。使用这种技术时，必须将 `InitialOwner` 标志设置为 `FALSE`；否则，可能难以保证哪一个进程获得初始占有权。

释放函数

BOOL ReleaseMutex(HANDLE hMutex)

参数

hMutex 互斥量句柄

返回值

如果成功，则返回 `TRUE`。如果不成功，则返回 `FALSE`。

描述

将互斥量的状态从非信号通知状态改为信号通知状态。仅当调用线程拥有互斥量的占有权时才发挥作用。如果调用线程不占有互斥量对象，则函数失败。

打开函数

HANDLE OpenMutex(DWORD fdwAccess, BOOL flInherit, LPTSTR lpzName)

参数

fdwAccess 值为 `SYNCHRONIZE` 或 `MUTEX_ALL_ACCESS`

flInherit 表明由这个进程创建的任何子进程是否应当继承这个互斥量对象的句柄。

lpzName 互斥量对象的一个零终止字符串名字。

返回值

如果发现指定互斥量对象，返回调用线程的句柄。如果没有发现，则返回 `NULL`。

描述

获得互斥量的句柄。如果互斥量对象通过指定名字找到，则创建进程相关句柄，它标识该互斥量，并返回调用线程的句柄。如果没有发现函数，则返回 `NULL`。

等待函数

DWORD WaitForSingleObject(HANDLE hObjects, DWORD dwTimeOut)

参数

hObjects 线程等待的核心对象的句柄。
dwTimeOut 线程等待对象发出信号的时长，以毫秒为单位。

返回值

返回一个返回码。返回码有：

WAIT_OBJECT_0 对象处于信号通知状态。
WAIT_TIMEOUT 在 dwTimeOut 毫秒内，对象不处于信号通知状态。
WAIT_ABANDONED 由于被放弃，对象为处于信号通知状态的互斥量。
WAIT_FAILED 发生一个错误。

描述

告诉系统线程正等待参数指定的核心对象被通知。如果对象没有在指定时间内被信号通知，系统将唤醒线程并让它继续执行。

DWORD WaitForMultipleObject(DWORD cObjects, CONST HANDLE lphObjects, BOOL bWaitAll, DWORD dwTimeOut)

参数

cObjects 被检查的对象数。
lphObjects 指向对象句柄的数组。数组可以包含不同类型对象的句柄：

对象类型	描 述
Thread	这个句柄由 CreateThread、CreateProcess 或 CreateRemoteThread 函数返回。线程终止时，线程对象的状态被信号通知
Semaphore	这个句柄由 CreateSemaphore 或 OpenSemaphore 函数返回。信号量对象的状态在它的计数大于 0 时被信号通知，等于 0 时为非信号通知。如果当前状态为信号通知，等待函数将把计数减少 1
Event	这个句柄由 CreateEvent 或 OpenEvent 函数返回。事件对象的状态被 SetEvent 或 PulseEvent 函数设置为信号通知
Mutex	这个句柄由 CreateMutex 或 OpenMutex 函数返回。互斥量对象在它不被任何线程占有时，它的状态为信号通知。等待函数向调用线程请求互斥量的占有权。授予占有权时，将互斥量的状态从信号通知改为非信号通知

续表

对象类型	描 述
Process	这个句柄由 <code>CreateProcess</code> 或 <code>OpenProcess</code> 函数返回。进程终止时, 进程对象的状态为信号通知
Change notification	这个句柄由 <code>FindFirstChangeNotification</code> 函数返回。
Console input	指定了 <code>CONIN\$</code> 时, 这个句柄由 <code>CreateFile</code> 函数返回, 或由 <code>GetStdHandle</code> 函数返回。当在控制台的输入缓冲中存在一个未读输入时, 对象的状态为信号通知。当控制台的输入缓冲为空时, 对象的状态为非信号通知

bWaitAll 决定线程是否应该等待列表中的一个或所有对象被信号通知。

dwTimeout 线程等待对象或对象被信号通知的时长 (以毫秒计)。

返回值

返回一个返回码。返回码为:

WAIT_OBJECT_0 to (WAIT_OBJECT_0 + cObjects - 1) 如果正等待所有对象, 这个值表示等待是成功的。如果线程正等待一个对象, 这个值表示数组中句柄的索引属于首先已被信号通知的对象。

WAIT_TIMEOUT 在指定时间内, 某个对象或某些对象没有进入信号通知状态。

WAIT_ABANDONED_0 to (WAIT_ABANDONED_0 + cObjects - 1) 如果线程正等待所有对象, 这个值表示等待是成功的, 而且至少有一个对象因为被放弃成为信号通知的互斥量。如果线程正等待一个对象, 这个值表示数组中句柄的索引属于由于被放弃而成为信号通知的互斥量对象。

WAIT_FAILED 发生一个错误。

描述

告诉系统该线程正等待几个核心对象成为信号通知, 或者等待列表中的一个对象被信号通知。如果这个对象或这些对象在指定时间内没有信号通知, 系统将唤醒线程并让它继续执行。如果 **bWaitAll** 参数为 **TRUE**, 线程将等待所有对象同时被信号通知。如果为 **FALSE**, 线程将等待一个对象被信号通知。

事件函数

创建/打开函数

HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpSa, BOOL fManualReset, BOOL fInitialState, LPCTSTR lpzEventName)

参数

lpSa SECURITY_ATTRIBUTES 结构的指针。

fManualReset 决定创建的对象是否为人工重置对象。如果为 TRUE，被创建对象为人工重置对象。否则，为 FALSE。

fInitialState 对象的初始状态。如果为 TRUE，所创建对象的初始状态为信号通知。否则，为 FALSE。

lpzEventName 如果被创建的对象有一个名字，它就是此名字字符串的指针。

返回值

返回事件对象的句柄。如果不成功，返回 NULL。

描述

创建一个命名或未命名事件对象。当事件对象的最后一个句柄被关闭时，它被销毁。

HANDLE OpenEvent(DWORD fdwAccess, BOOL fInherit, LPCTSTR lpzEventName)

参数

fdwAccess 指定访问事件对象的请求。这个参数可以为以下值的任意组合：

EVENT_ALL_ACCESS 为事件对象指定所有可能的访问标志。

EVENT_MODIFY_STATE 允许在设置事件和重置事件函数中使用事件句柄来修改事件的状态。

SYNCHRONIZE 允许在任何等待函数中使用事件句柄来等待事件的状态成为信号通知。

fInherit 指定返回句柄是否可继承。

lpzEventName 指向一个命名被打开事件的 null 终止字符串。

返回值

如果成功，返回事件对象的句柄。如果不成功，返回 NULL。

描述

返回现有命名事件对象的一个句柄。打开事件函数允许多个进程打开同一个事件对象的多个句柄。仅当某些进程已经使用事件创建函数创建了事件时，函数才成功。调用进程可以在要求事件对象的任何函数中使用这个返回句柄。当进程终止时，系统自动关闭这个句柄。当最后一个句柄被关闭时，销毁这个事件对象。

信号通知函数

BOOL SetEvent(HANDLE hEvent)

参数

hEvent 标识事件对象。

返回值

如果成功，返回 TRUE。如果不成功，返回 FALSE。

描述

将指定事件对象的状态设置为信号通知。

重置函数

BOOL ResetEvent(HANDLE hEvent)

参数

hEvent 标识事件对象。

返回值

如果成功，返回 TRUE。如果不成功，返回 FALSE。

描述

将指定事件对象的状态设置为非信号通知。这个函数主要用于人工重置事件对象。它必须显式设置为非信号通知状态。释放一个等待线程后，自动重置事件对象自动把状态从信号通知改为非信号通知。事件对象的状态一直保持非信号通知，直到它被设置事件函数或脉冲事件函数设置成信号通知为止。

信号量函数

创建函数

HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpsa, LONG cSemInitial, LONG cSemMax, LPCSTR lpszSemName)

参数

lpsa SECURITY_ATTRIBUTES 结构的指针。

cSemInitial 信号量的初始计数。

cSemMax 信号量的最大计数。

lpszSemName 信号量名字。

返回值

如果成功, 返回信号量对象的句柄。如果不成功, 返回 NULL。

描述

创建一个具有参数 *cSemMax* 的最大资源计数的信号量。当计数大于 0 时, 信号量对象的状态为信号通知, 当计数等于 0 时, 信号量对象的状态为非信号通知。*cSemInitial* 参数指定初始计数。由于信号量处于信号通知状态而释放一个等待线程时, 信号量的计数减去 1。当它的最后一个句柄被关闭时, 销毁这个信号量对象。

HANDLE OpenSemaphore(DWORD fdwAccess, BOOL flnherit, LPTSTR lpzName)

参数

fdwAccess 值为 SYNCHRONIZE 或 MUTEX_ALL_ACCESS。
flnherit 表明这个进程创建的任何子进程是否应该继承该信号量对象的句柄。
lpzName 信号量对象的零终止字符串名字。

返回值

如果发现了指定信号量对象, 返回调用线程句柄。如果没有发现, 返回 NULL。

描述

获取信号量的句柄。如果通过指定名字找到了信号量对象, 创建一个相对进程的句柄, 它标识该互斥量并返回调用线程的这个句柄。如果没有发现, 则返回 NULL。

释放函数

BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG cRelease, LPLONG lplPrevious)

参数

hSemaphore 信号量句柄。
cRelease 表明应该释放多少信号量。
lplPrevious 指向信号量的前一个计数。

返回值

如果成功, 返回 TRUE。如果不成功, 返回 FALSE。

描述

增加指定信号量对象的计数为一定量。当计数大于 0 时, 信号量对象的状态为信号通知, 当计数等于 0 时, 信号量对象的状态为非信号通知。等待线程由于信号量的信号通知状态而被释放时, 信号量的计数减少 1。这个函数也可以在应用程序的初始期间使用。应用程序可以用初始计数 0 来

创建信号量。这将信号量的状态设置为非信号通知，并且阻塞所有的线程，不能访问保护资源。当应用程序完成初始化后，它使用这个函数来增加计数到最大值，允许对保护资源的正常访问。

临界区函数

VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection)

参数

lpCriticalSection CRITICAL_SECTION 结构的地址。

返回值

无返回值。

描述

初始化结构的成员。在调用 EnterCriticalSection()前调用它。

VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)

参数

lpCriticalSection CRITICAL_SECTION 结构的地址。

返回值

无返回值。

描述

等待指定临界区对象的占有权。授予调用线程占有权时，函数返回。要允许访问共享资源，每个线程在执行任何访问保护资源的代码段前，必须调用这个函数来请求临界区的占有权。如果另一个当前线程占有临界区对象，进入临界区函数无限期阻塞调用线程的执行，直到其他线程释放占有权。如果临界区对象当前未被占有，系统授予占有权给请求的线程。

VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection)

参数

lpCriticalSection 指向临界区对象。

返回值

无返回值。

描述

释放指定临界区对象的占有权。线程使用这个函数来获取临界区对象的占有权。如果线程没

有指定临界区对象的占有权时调用这个函数，则发生错误，可能导致另一个使用进入临界区函数的线程无限期等待。

管道函数

匿名管道创建函数

BOOL CreatePipe(PHANDLE *phRead*, PHANDLE *phWrite*, LPSECURITY_ATTRIBUTES *lpsa*, DWORD *cbPipe*)

参数

<i>phRead</i>	指向匿名管道读末端句柄的指针。
<i>phWrite</i>	指向匿名管道的写末端句柄的指针。
<i>lpsa</i>	SECURITY_ATTRIBUTES 结构的指针。
<i>cbPipe</i>	指定为管道保留字节数的变量指针。

返回值

如果成功，返回 TRUE。如果不成功，返回 FALSE。

描述

创建一个匿名管道。这个函数给存储缓冲分配指定管道大小。这个函数还创建进程在后面调用读和写文件函数中用于读和写缓冲的句柄。当进程使用写文件函数来写入匿名管道，写操作直到所有字节写入后才完成。如果在写入所有字节前管道已满，写文件函数将一直等到另一个进程或线程使用读文件函数腾出更多可用缓冲空间后才会返回。

如果安全属性结构的指针为 NULL，管道的句柄不会被继承。如果为管道保留的字节数指针为 NULL，系统将使用缺省值。

命名管道创建函数

HANDLE CreateNamedPipe(LPCTSTR *lpName*, DWORD *dwOpenMode*, DWORD *dwPipeMode*, DWORD *nMaxInstances*, DWORD *nOutBufferSize*, DWORD *nInBufferSize*, DWORD *nDefaultTimeout*, LPSECURITY_ATTRIBUTES *lpSecurityAttributes*)

参数

<i>lpName</i>	指向标识管道的 null 终止字符串。
<i>dwOpenMode</i>	指定管道的访问模式、重叠模式(overlapped mode)、直写模式(write-through mode)以及管道句柄的安全访问模式。这个参数指定以下几种管道访问模式标志： PIPE_ACCESS_DUPLEX 管道为双向；服务器和客户进程都可以读和写管道。

PIPE_ACCESS_INBOUND 管道中的数据流只从客户进入服务器。服务器也有等价的访问模式。

PIPE_ACCESS_OUTBOUND 管道中的数据流只从服务器进入。服务器也有等价的访问模式。这个参数也可以包括以下两个参数之一，或者同时包括两个参数：

FILE_FLAG_WRITE_THROUGH 允许直写模式。

FILE_FLAG_OVERLAPPED 允许重叠模式。对于需要大量时间来完成的读、写和连接运算，这个模式允许执行这些运算的函数立即返回。这个参数可以包括以下安全访问模式标志的任意组合。这些模式对于同样管道的不同实例有效：

WRITE_DAC 调用者可以写访问命名管道的任意访问控制列表(access control list, ACL)。

WRITE_OWNER 调用者可以写访问命名管道的占有者。

ACCESS_SYSTEM_SECURITY 调用者可以写访问命名管道的系统 ACL。

dwPipeMode 指定管道句柄的类型、读和等待模式。有以下几种模式：

PIPE_TYPE_BYTE 数据作为字节流写入管道。这个模式不能与 **PIPE_READMODE_MESSAGE** 一起使用。

PIPE_TYPE_MESSAGE 数据作为消息流写入管道。这个模式可以与 **PIPE_READMODE_MESSAGE** 或 **PIPE_READMODE_BYTE** 一起使用。可以指定以下读模式标志之一：

PIPE_READMODE_BYTE 数据作为字节流读取管道。这个模式能与 **PIPE_TYPE_MESSAGE** 或 **PIPE_TYPE_BYTE** 一起使用。

PIPE_READMODE_MESSAGE 数据作为消息流读取管道。这个模式可以与 **PIPE_TYPE_MESSAGE** 一起使用。可以指定以下等待模式标志之一：

PIPE_WAIT 允许阻塞模式。使用这个模式意味着无限期等待客户进程执行某个动作。

PIPE_NOWAIT 允许非阻塞模式。读、读和文件函数以及连接命名管道函数总是立即返回。

nMaxInstances 指定这个管道可以创建的最大实例数量。

nOutBufferSize 指定为输出缓冲保留的字节数。

nInBufferSize 指定为输入缓冲保留的字节数。

nDefaultTimeOut 如果等待命名管道函数指定 **NMPWAIT_USE_DEFAULT_WAIT**，它指定缺省的时间耗尽值（以毫秒计）。

lpSecurityAttribute 指向 **SECURITY_ATTRIBUTES** 结构，它指定管道的安全性属性。

返回值

返回句柄的值给命名管道实例的服务器终端。如果不成功，返回 **INVALID_HANDLE_VALUE**。

描述

创建一个命名管道并返回一个管道句柄的管道。命名管道服务器进程使用这些函数来创建指

定命名管道的第一个实例，并建立它的基本属性，或者创建现有命名管道的新实例。

当创建命名管道的一个实例时，用户必须能够访问命名管道对象。如果创建新命名管道，指定安全性属性结构的访问控制列表（ACL）定义了该命名管道的访问控制。所有命名管道的实例必须指定具有如下特点的不同管道类型：字节或消息管道；作为双向、入管或出管的管道访问；实例计数以及时间耗尽值。如果使用了不同的值，这个函数将失败，而且错误函数返回 `ERROR_ACCESS_DENIED`。每个命名管道终端的实际缓冲大小将为系统缺省值，系统最小值或最大值，或者近似为邻近分配边界的指定大小。当命名管道实例的最后一个句柄被关闭时，删除命名管道的实例。

命名管道连接/调用/事务函数

BOOL ConnectNamedPipe(HANDLE *hNamedPipe*, LPOVERLAPPED *lpOverlapped*)

参数

hNamedPipe 标识命名管道实例的服务器终端。
lpOverlapped 指向一个 OVERLAPPED 结构。

返回值

如果成功，返回 `TRUE`。如果不成功，返回 `FALSE`。

描述

允许命名管道服务器进程等待客户进程连接到命名管道的某个实例上。客户进程通过调用这个函数或创建文件函数连接到命名管道。命名管道服务器进程可以用新创建的管道实例使用这个函数。它也可以通过一个先前连接到另一个客户进程的命名管道实例来使用。在这里，将句柄重新连接到新客户前，服务器进程必须首先调用断开连接命名管道函数与前一个客户断开句柄连接。如果不这样做，函数将返回 `FALSE`。如果前面的函数已经关闭了它的句柄，错误函数将返回 `ERROR_NO_DATA`，或者没有关闭它的句柄，则返回 `ERROR_PIPE_CONNECTED`。

这个函数的行为取决于管道句柄的等待模式是否设置为阻塞或非阻塞，而且函数是否设置为同步执行或处于重叠模式。当创建命名管道时，服务器初始指定管道的等待模式。使用设置命名管道句柄状态的函数可以更改它。如果在创建管道句柄时指定了 `FILE_FLAG_OVERLAPPED`，而且 `lpOverlapped` 参数不是 `NULL`，函数将作为重叠运算进行执行。如果不是这样，则函数同步执行。如果使用重叠运算，则由 `lpOverlapped` 指向的 `OVERLAPPED` 结构必须包含人工重置事件对象的句柄。

如果 `lpOverlapped` 参数为 `NULL`，系统将试图同步执行运算。如果尝试成功，直到连接到客户或发生错误时，函数才同步执行。如果函数调用后客户连接，函数将返回 `TRUE`。如果调用函数前，客户连接到管道，函数将返回 `FALSE`，而且错误函数将返回 `ERROR_PIPE_CONNECTED`。如果试图同步执行运算失败，函数可能以不可预测的方式失败。

如果指定的管道句柄处于非阻塞模式，这个函数将总是立即返回。在非阻塞状态，这个函数将在第一次对一个与先前客户断开连接的管道实例调用时返回 **TRUE**。这说明这个管道可以连接到新客户进程。在所有其他场合，当管道句柄处于非阻塞状态，这个函数将返回 **FALSE**。在这些场合，错误函数将返回如下值：

ERROR_PIPE_LISTENING 无客户连接。

ERROR_PIPE_CONNECTED 一个客户连接。

ERROR_NO_DATA 先前客户已经关闭它的管道句柄，但服务器没有断开连接。

ERROR_PIPE_CONNECTED 客户与服务之间存在稳定连接。

BOOL CallNamedPipe(LPCTSTR *lpzPipeName*, LPVOID *lpvWriteBuf*, DWORD *cbWriteBuf*, LPVOID *lpvReadBuf*, DWORD *cbReadBuf*, LPDWORD *lpcbRead*, DWORD *dwTimeout*)

参数

lpzPipeName 指向指定管道名字的 null 终止字符串。

lpvWriteBuf 指向包含写入管道数据的缓冲。

cbWriteBuf 指定输入缓冲的字节大小。

lpvReadBuf 当从管道读取数据时，指向输入缓冲。

cbReadBuf 指定输入缓冲的字节大小。

lpcbRead 指向一个 32 位变量，它接收从管道读取的字节数。

dwTimeout 指定等待命名管道变成可用的时间限制，以毫秒为单位。除了数字值外，还可以指定如下特殊值：

NMPWAIT_NOWAIT 不等待命名管道。如果命名管道不可用，函数将返回一个错误。

NMPWAIT_WAIT_FOREVER 无限地等待命名管道。

NMPWAIT_USE_DEFAULT_WAIT 使用指定的缺省时间耗尽。

返回值

如果成功，返回 **TRUE**。如果不成功，返回 **FALSE**。

描述

连接到消息管道并读写管道，然后关闭管道。这个函数等待决定管道的某个实例是否可用。如果由服务器写入管道的消息长于输入缓冲，函数将返回 **FALSE**，而且错误函数将返回 **ERROR_MORE_DATA**。消息的剩余部分被删除，因为这个函数在返回前关闭了管道的句柄。如果管道不是一个消息管道，这个函数将失败。

BOOL TransactNamedPipe(HANDLE *hNamedPipe*, LPVOID *lpvWriteBuf*, DWORD *cbWriteBuf*, LPVOID *lpvReadBuf*, DWORD *cbReadBuf*, LPDWORD *lpcbRead*, LPOVERLAPPED *lpo*)

参数

<i>hNamedPipe</i>	标识命名管道。
<i>lpvWriteBuf</i>	指向输入缓冲，它包含写入管道的数据。
<i>cbWriteBuf</i>	指定输入缓冲的字节大小。
<i>lpvReadBuf</i>	指向输入缓冲，它接收读取管道的数据。
<i>cbReadBuf</i>	指定输入缓冲的字节大小。
<i>lpcbRead</i>	指向接收从管道读取的字节数的变量。
<i>lpo</i>	指向一个 OVERLAPPED 结构。

返回值

如果成功，返回 TRUE。如果不成功，返回 FALSE。

描述

从指定的命名管道写入或读取消息。如果服务器没有作为消息管道创建这个管道，或者管道句柄不处于消息读模式，这个函数将失败。管道句柄的等待模式，不管它是阻塞或非阻塞，都对这个函数的行为无影响。直到数据写入输出缓冲后，函数才能完成。如果读取的消息长于输入缓冲，则函数返回 FALSE，而且错误函数将返回 ERROR_MORE_DATA。

创建命名管道时，如果指向 OVERLAPPED 结构的参数为 NULL，或者没有指定 FILE_FLAG_OVERLAPPED 标志，函数直到操作完成后才能返回。创建句柄时，如果参数不是 NULL，而且没有指定 FILE_FLAG_OVERLAPPED，这个函数作为一个重叠操作来执行。OVERLAPPED 结构应当包含人工重置事件对象。如果操作不能立即完成，这个函数将返回 FALSE，而且错误函数将返回 ERROR_IO_PENDING。在这种情况下，函数返回前，事件对象被设置为非信号通知状态。当事务完成时，事件对象被设置成信号通知状态。

命名管道断开连接函数

BOOL DisconnectNamedPipe(HANDLE *hNamedPipe*)

参数

<i>hNamedPipe</i>	标识命名管道的一个实例
--------------------------	-------------

返回值

如果成功，返回 TRUE。如果不成功，返回 FALSE。

描述

断开连接命名管道实例的服务器终端与客户进程。如果命名管道的客户终端打开，DisconnectNamedPipe 函数强迫命名管道终端的关闭。当客户下次试图访问这个管道时，它将收到

一个错误。被 `DisconnectNamedPipe` 强迫离开管道的客户必须仍然使用 `CloseHandle` 函数来关闭管道终端。

当服务器进程断开管道实例时，丢掉管道中的任何未读数据。在断开之前，服务器可以调用 `FlushFileBuffers` 函数确保数据不丢失。直到客户进程读取所有数据后，`FlushFileBuffers` 函数才返回。在使用 `ConnectNamedPipe` 函数将句柄连接到另一个客户前，服务器进程必须调用 `DisconnectNamedPipe` 断开管道句柄与它的前一个客户的连接。

命名管道查询函数

BOOL `GetNamedPipeHandleState`(**HANDLE** *hNamedPipe*, **LPDWORD** *lpdwState*, **LPDWORD** *lpdwCurInstances*, **LPDWORD** *lpdwMaxCollect*, **LPDWORD** *lpdwCollectTimeout*, **LPTSTR** *lpszUser*, **DWORD** *cchMaxUser*)

参数

hNamedPipe 标识查询信息所针对的命名管道。

lpdwState 指向一个 32 位变量，它表明句柄的当前状态。如果需要，这个参数可以为 NULL。可以指定如下任何一个值，也可以同时指定两个值：

PIPE_NOWAIT 管道句柄处于非阻塞模式。如果没有指定，管道句柄处于阻塞模式。

PIPE_READMODE_MESSAGE 管道句柄处于消息读模式。如果没有指定，管道句柄处于字节读模式。

lpdwCurInstances 指向某个变量，它接收当前管道的实例数。

lpdwMaxCollect 指向某个变量，它接收传输到服务器之前，客户计算机上搜集的最大字节数。

lpdwCollectTimeout 指向某个变量，它接收在网络上传输信息前，远程命名管道应当等待的最长时间，以毫秒为单位。

lpszUser 指向一个缓冲，这个缓冲接收包含客户应用的用户名字符串的 null 终止字符串。

cchMaxUser 指定缓冲器的大小，这个缓冲器由 *lpszUser* 参数指定，以字节为单位。

返回值

如果成功，返回 TRUE。如果不成功，返回 FALSE。

描述

读取指定命名管道的信息。在命名管道实例的生存期间，这个信息可能不同。即使所有的传递指针都为 NULL，这个函数也将成功返回。如果表示搜集最大字节数的参数具有 NULL 值，则命名管道的服务器终端或客户与服务器进程的指定管道句柄位于同一台计算机上。如果不需要这个信息，参数可以为 NULL。如果指定通过网络传输信息前，远程命名管道必须等待的最长时间参数为 NULL，则命名管道的服务器终端或客户与服务器进程的指定管道句柄位于同一台计算机上。如果不需要这个信息，参数可以为 NULL。如果指向接收 null 终止字符串（包含客户应用的

用户名字符串) 的缓冲的参数为 NULL, 则指定管道句柄就是针对命名管道的客户终端。如果不需要这个信息, 参数可以为 NULL。

BOOL GetNamedPipeInfo(HANDLE *hNamedPipe*, LPDWORD *lpdwType*, LPDWORD *lpcbOutBuf*, LPDWORD *lpcbInBuf*, LPDWORD *lpcMaxInstances*)

参数

hNamedPipe 标识命名管道实例。

lpdwType 指向一个变量, 这个变量表明命名管道的类型, 而且指定句柄是否针对命名管道的服务器或客户终端。如果不需要这个信息, 参数可以为 NULL。可以指定如下任何一个值, 也可以同时指定:

PIPE_SERVER_END 句柄引用命名管道实例的服务器终端。如果没有指定这个值, 句柄引用命名管道实例的客户终端。

PIPE_TYPE_MESSAGE 命名管道为一个消息管道。如果没有指定这个值, 管道就是字节管道。

lpcbOutBuf 指向一个变量, 这个变量接收用于传出数据的缓冲大小。以字节为单位。

lpcbInBuf 指向一个变量, 这个变量接收用于传入数据的缓冲大小。以字节为单位。

lpcMaxInstances 指向一个变量, 这个变量接收可创建管道实例的最大数。

返回值

如果成功, 返回 TRUE。如果不成功, 返回 FALSE。

描述

读取指定命名管道的相关信息。在命名管道实例的生存期间, 返回的信息可以改变。如果接收传出数据缓冲大小的参数值为 0, 则缓冲按需要分配。如果不需要这个信息, 这个参数可以为 NULL。如果接收传入数据缓冲大小的参数值为 0, 则缓冲按需要分配。如果不需要这个信息, 这个参数可以为 NULL。如果接收管道实例最大数的参数设置为 PIPE_UNLIMITED_INSTANCES, 则管道实例数仅受可用系统资源的限制。如果不需要这个信息, 参数可以为 NULL。

设置函数

BOOL SetNamedPipeHandleState(HANDLE *hNamedPipe*, LPDWORD *lpdwMode*, LPDWORD *lpcbMaxCollect*, LPDWORD *lpdwCollectDataTimeout*)

参数

hNamedPipe 标识命名管道实例。

lpdwMode 指向一个提供新模式的变量。以下为读模式:

PIPE_READMODE_BYTE 数据作为字节流从管道读取。如果没有指定读模式标志, 这个模式就是缺省模式。

PIPE_READMODE_MESSAGE 数据作为消息流从管道读取。以下为等待模式:

PIPE_WAIT 允许阻塞模式。如果没有指定等待模式标志, 这个模式就是缺省模式。当在读文件、写文件或连接命名管道函数中指定一个阻塞模式管道句柄时, 这些操作直到有数据可读、所有数据被写入或者连接到客户后才能完成。使用这些模式可能意味着无限期等待客户进程执行某个动作。

PIPE_NOWAIT 允许非阻塞模式。处于这个模式下, 读文件、写文件以及连接命名管道函数总是立即返回。

lpchMaxCollect 指向一个变量, 这个变量指定传输到服务器前, 在客户计算机上搜集的最大字节数。

lpdwCollectDataTimeout 指向一个变量, 这个变量指定在远程命名管道在网络上传输信息前, 可以等待的最长时间, 以毫秒为单位。

返回值

如果成功, 返回 **TRUE**。如果不成功, 返回 **FALSE**。

描述

设置指定命名管道的读模式和阻塞模式。如果句柄是针对命名管道的客户终端, 以及命名管道服务器进程位于同一台远程计算机上, 这个函数也可以用于局部缓冲。模式参数是读模式与等待模式标志的组合。如果没有设置这个模式, 这个参数可以为 **NULL**。如果指定传输到服务器之前客户计算机上搜集的最大字节数的参数为 **NULL**, 则命名管道的服务器终端或客户与服务器进程的指定管道句柄位于相同计算机上。如果创建句柄时, 客户进程指定了 **FILE_FLAG_WRITE_THROUGH** 标志, 则忽略这个参数。如果没有设置搜集计数, 这个参数可以为 **NULL**。如果指定远程命名管道在网络上传输信息前最长等待时间的参数为 **NULL**, 则命名管道的服务器终端或客户与服务器进程的管道句柄位于相同的计算上。如果创建句柄时, 客户进程指定了 **FILE_FLAG_WRITE_THROUGH** 标志, 则忽略这个参数。如果没有设置搜集计数, 这个参数可以为 **NULL**。

命名管道等待函数

BOOL WaitNamedPipe(LPCTSTR lpzPipeName, DWORD dwTimeout)

参数

lpzPipeName 指向一个 null 终止字符串, 它指定了命名管道的名字。

dwTimeout 指定函数等待命名管道实例变成可用的时间, 以毫秒为单位。如果不指定毫秒数, 也可以使用如下值之一:

NMPWAIT_USE_DEFAULT_WAIT 时间耗尽段是创建命名管道时, 由服务器进程指定的缺省值。

NMPWAIT_WAIT_FOREVER 函数将等到命名管道实例变成可用时才返回。

返回值

如果时间段耗尽前管道实例可用,则返回 **TRUE**。如果成功,也返回 **TRUE**。否则,返回 **FALSE**。

描述

等到时间段过期,或者指定命名管道的某个实例可用于连接。如果不存在指定命名管道的实例,则等待命名管道函数将立即返回,而不管时间耗尽值为多少。如果函数成功,进程应该使用创建文件函数来打开命名管道的一个句柄。返回值 **TRUE** 表示至少管道的一个实例可用。一旦服务器关闭了命名管道的实例,或者由另一个客户打开此实例,针对该管道对创建文件函数的后续调用可能失败。

附录 E OS/2 线程管理函数

线程属性

信 息	描 述
<i>结构 1</i>	
exception_chain	线程异常处理器头的指针
stack_base	线程堆栈基的指针
stack_limit	线程堆栈尾的指针
more_info	TIB2 结构的指针, 这个结构包含与线程相关的更多信息
version	TIB 结构的版本号
ordinal	线程序号
<i>结构 2</i>	
tid	当前线程的线程标识符
priority	当前线程的优先权
version	TIB2 结构的版本号
count	由 OS2 使用
forceflag	由 OS2 使用

线程属性函数

APIRET DosGetInfoBlocks(PPTIB ptib, PPIB ppib)

参数

ptib 线程信息块的指针地址。
ppib 进程信息块的指针地址。

返回值

如果成功，返回线程和进程信息块的地址。

描述

返回当前线程的线程信息块地址。它还返回当前进程的进程信息块地址。线程信息块包含以下信息：

struct _TIB

PVOID exception_chain 线程异常处理器头的指针。
 PVOID stack_base 线程堆栈基的指针。
 PVOID stack_limit 线程堆栈尾的指针。
 PTIB2 more_info TIB2 结构的指针，这个结构包含与线程相关的更多信息。
 ULONG version TIB 结构的版本号。
 PVOID ordinal 线程序号。

struct _TIB2

ULONG tid 当前线程的线程标识符。
 ULONG priority 当前线程的优先权。
 ULONG version TIB2 结构的版本号。
 USHORT count 由 OS2 使用。
 USHORT forceflag 由 OS2 使用。

线程创建

APIRET DosCreateThread(PTID ThreadID, PFTHREAD ThreadAddr, ULONG ThreadArg, ULONG ThreadFlags, ULONG StackSize)

参数

ThreadID 线程标识符。
 ThreadAddr 线程函数指针。

ThreadArg 传递给线程的参数。
 ThreadFlags 创建标志。
 StackSize 线程堆栈的大小。

返回值

如果成功, 返回一个返回码以及线程标识符。返回码有:

NO_ERROR
 ERROR_NOT_ENOUGH_MEMORY
 ERROR_INTERRUPT
 ERROR_PROTECTION_VIOLATION
 ERROR_MAX_THRDS_REACHED

描述

在当前进程中创建一个异步执行线程。标志参数决定线程是否以挂起状态来创建。如果位 1 的值为 1, 则挂起线程, 而且直到另一个线程恢复该线程后才返回。如果位 1 的值为 0, 则立即执行线程。堆栈大小的最小值为 4096 字节。如果创建标志的位 1 值为 0, 操作系统将使用缺省方法来初始化线程堆栈。如果位 1 的值为 1, 预分配线程整个堆栈的内存。

线程终止函数

APIRET DosExit(ULONG terminate_action, ULONG exit_code)

参数

terminate_action 指定终止的对象。如果值为 0, 则终止当前线程。
 exit_code 完成状态。

返回值

返回退出码。

描述

终止一个线程或进程。当终止时, 它不返回到调用者, 只是停止运行。

APIRET DosKillThread(TID ThreadID)

参数

ThreadID 被终止线程的线程标识符。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_BUSY

ERROR_INVALID_THREADID

描述

在当前进程中终止一个线程。它强迫线程终止，但不导致整个进程的终止。它不等待终止线程完成终止进程而返回到调用线程。

APIRET DosWaitThread(PTID ThreadID, ULONG WaitOption)

参数

ThreadID 线程标识符。

WaitOption 决定调用线程是否等待到线程终止，还是立即返回。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_THREAD_NOT_TERMINATED

ERROR_INVALID_THREADID

如果线程标识符为调用线程的标识符，或者该线程为线程 1，返回 ERROR_INVALID_THREADID。

描述

将当前线程置入等待状态，直到当前进程中的另一个线程终止为止。

由终止线程返回线程标识符。如果指定线程为调用线程或主线程，函数将返回 ERROR_INVALID_THREADID。

线程管理函数

优先权函数

APIRET DosSetPriority(ULONG Scope, ULONG PriorityClass, LONG PriorityDelta, ULONG ID)

参数

Scope 决定是否改变一个线程或进程的优先权。

PriorityClass 优先类。
 PriorityDelta 添加到当前级别上的增量。
 ID 如果作用域值为 2，则为线程标识符。

返回值

返回一个返回码。返回码如下：

NO_ERROR
 ERROR_INVALID_PROCID
 ERROR_INVALID_PDELTA
 ERROR_NOT_DESCENDANT
 ERROR_INVALID_PCLASS
 ERROR_INVALID_SCOPE
 ERROR_INVALID_THREADID

描述

更改线程的基优先权。线程的优先权通过在现有优先权级别上添加某个值来更改，这个值称做增量（delta）。优先权的修改是相对于当前优先级的。如果增量为正，则优先级提高。如果增量为负，则优先级降低。增量值限制于基于进程的当前优先类有效范围。如果优先类与优先级同时更改，则增量相对为 0。当函数通过一个类定义调用，但增量无指定值时，基优先级将为 0。

作用域参数指定优先权更改的范围。可以取以下值：

- 0 指定进程及所有线程。
- 1 当前进程及所有后代。
- 2 当前进程中的单个线程。

优先类参数可以取以下值之一：

- 0 不更改类。
- 1 空闲时间优先类。
- 2 常规优先类。
- 3 时间敏感性优先类。
- 4 固定高优先类。

优先权增量参数可以取-31 到 31 的某个值。

挂起/恢复函数

APIRET DosSuspendThread(TID ThreadID)

参数

ThreadID 被挂起线程的线程标识符。

返回值

返回一个返回码。返回码如下：

NO_ERROR
ERROR_NOT_FROZEN
ERROR_INVALID_THREADID

描述

在当前进程中，临时挂起另一个线程的执行，直到调用恢复函数为止。线程可能不会被立即挂起，因为它可能锁定某资源，必须首先释放它。只有调用恢复函数时，线程可以继续执行。

APIRET DosResumeThread(TID ThreadID)

参数

ThreadID 被恢复线程的线程标识符。

返回值

返回一个返回码。返回码如下：

NO_ERROR
ERROR_NOT_FROZEN
ERROR_INVALID_THREADID

如果线程不处于挂起状态，返回 ERROR_NOT_FROZEN。

描述

重新启动一个被挂起的线程。如果线程没有被挂起，返回 ERROR_NOT_FROZEN 返回码。

互斥量函数

创建函数

APIRET DosCreateMutexSem(PSZ Name, PHMTX phmtx, ULONG flAttr, BOOL fState)

参数

Name 信号量 ASCII 名字的指针。
phmtx 互斥信号量句柄的指针。
flAttr 指定信号量属性的一套标志。
fState 信号量的初始状态。如果：
 0 (FALSE) 信号量的初始状态为未被占有 (unowned)。

1 (TRUE) 信号量的初始状态为被占有 (owned)。

返回值

返回一个返回码。返回码如下：

NO_ERROR
 ERROR_NOT_ENOUGH_MEMORY
 ERROR_INVALID_PARAMETER
 ERROR_INVALID_NAME
 ERROR_DUPLICATE_NAME
 ERROR_TOO_MANY_HANDLES

如果文件系统没有确认信号量名字，不包括前缀\SEM32\，超过了最多的 255 个字符，则返回 ERROR_INVALID_NAME。如果信号量已经存在，则返回 ERROR_DUPLICATE_NAME。

描述

创建一个互斥信号量，并对当前进程中的所有线程打开它。当创建了互斥量时，设置一个标志决定互斥量是否被占有或未被占有。如果互斥量被一个线程占有，其他请求该互斥量的线程将阻塞。如果未被占有，则请求该互斥量的所有线程立即得到占有权。如果调用线程将互斥量的初始状态设置为被占有，一旦创建互斥量后，它就被占有。如果互斥量未被占有，则进程中的任何线程都可以请求占有权。

释放/关闭函数

OS2

APIRET DosReleaseMutexSem(HMTX hmtx)

参数

hmtx 互斥信号量句柄。

返回值

返回一个返回码。返回码如下：

NO_ERROR
 ERROR_INVALID_HANDLE
 ERROR_NOT_OWNER

描述

释放互斥信号量的占有权。占有互斥量的线程调用这个函数。每次调用此函数将减小请求计数，互斥量的这个计数由操作系统维护。直到请求计数降为 0，这个互斥量才真正释放给另一个

线程。

APIRET DosCloseMutexSem(HTMX htmx)

参数

htmx 互斥信号量的句柄。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_INVALID_HANDLE

ERROR_SEM_BUSY

如果互斥量被进程中的另一个线程占有，通过请求该互斥量阻塞另一个线程，或者线程尝试关闭仍然被占有的互斥量，则返回 ERROR_SEM_BUSY。

描述

关闭互斥量。当进程不再需要访问互斥量时，必须关闭它。如果进程没有关闭互斥量而终止，互斥量将由操作系统关闭。

每次调用这个函数将减少互斥量计数。当创建互斥量时，它被初始化为 1。当计数为 0 时，操作系统删除互斥量。这称作终结关闭。对于以下情况不发生终结关闭：

 互斥量被进程中的另一个线程占有。

 进程中的另一个线程被一个互斥量请求阻塞。

打开/查询函数

APIRET DosOpenMutexSem(PSZ Name, PHMTX phmtx)

参数

Name 打开的信号量 ASCII 名字的指针。

phmtx 互斥信号量句柄的指针。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_INVALID_HANDLE

ERROR_NOT_ENOUGH_MEMORY

ERROR_INVALID_PARAMETER

ERROR_SEM_OWNER_DIED

ERROR_INVALID_NAME

ERROR_SEM_NOT_FOUND
ERROR_TOO_MANY_OPENS

如果进程占有一个互斥量，而且终止时没有释放互斥量，请求该互斥量的所有线程都将收到一个返回码 **ERROR_SEM_OWNER_DIED**。

描述

打开一个互斥量。其他进程中的线程使用它来访问互斥量。一旦被另一个进程中的线程打开，它就对该进程中的其他所有线程开放。这个函数提供了访问互斥量的权限，但没有提供占有权。当进程不再要求这个互斥量进，必须关闭互斥量。如果进程终止时没有关闭互斥量，则由操作系统关闭互斥量。只要函数用于相同的互斥量，互斥量计数就增加。创建互斥量时，计数初始化为 1。当关闭互斥量时，计数减少。若计数值为 0，则从系统中删除这个互斥量。

如果进程占有一个互斥量，并在终止时没有释放它，任何请求该互斥量的线程都将收到一个返回码 **ERROR_SEM_OWNER_DIED**。这个互斥量仍然对调用进程打开。

APIRET DosQueryMutexSem(HTMX htmx, PPID ppidOwner, PTID ptidOwner, PULONG pulCount)

参数

htmx 互斥信号量的句柄。
ppidOwner 占有互斥量的进程标识符。
ptidOwner 占有互斥量的线程标识符。
pulCount 请求互斥量的数量。

返回值

返回互斥量占有者的进程和线程标识符，以及请求该互斥量的数量。返回一个返回码。返回码如下：

NO_ERROR
ERROR_INVALID_HANDLE
ERROR_INVALID_PARAMETER
ERROR_SEM_OWNER_DIED

如果占有互斥量的进程终止时没有关闭互斥量，则返回 **ERROR_SEM_OWNER_DIED**。

描述

决定互斥量的占有者并获取请求的数量。如果互斥量没有被占有，这个值为 0。这个函数可以被创建互斥量的进程中的所有线程调用。其他进程的线程如果通过调用打开互斥量函数首先获得了访问互斥量的权限，它们可以调用这个函数。如果占有互斥量的进程终止时没有关闭互斥量，则函数返回 **ERROR_SEM_OWNER_DIED**。参数将包含与终止的占有进程相关的信息。

请求函数

OS2

APIRET DosRequestMutexSem(HTMX htmx, ULONG TimeOut)

参数

htmx 互斥信号量的句柄。

TimeOut 决定线程阻塞于请求的时间长短。这个参数的值可以为以下之一：

SEM_IMMEDIATE_RETURN 不阻塞线程而返回。线程可以执行其他操作，如果它仍然请求访问受保护资源，可在晚些时候再次调用函数。

SEM_INDEFINITE_WAIT 线程无限期等待。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_INVALID_HANDLE

ERROR_INTERRUPT

ERROR_TOO_MANY_SEM_REQUESTS

ERROR_SEM_OWNER_DIED

ERROR_TIMEOUT

如果在线程获得互斥量的占有权之前达到了时间限制，则返回 **ERROR_TIMEOUT**。如果线程被外部事件取消阻塞的同时，它正等待互斥量，则返回 **ERROR_INTERRUPT**。如果试图请求互斥量比允许的时间要多，则返回 **ERROR_TOO_MANY_REQUESTS**。

描述

请求互斥量的占有权。这个函数可以被创建互斥量的进程中任意线程来调用。其他进程的线程如果通过调用打开互斥信号量函数首先获得了访问互斥量的权限，它们也可以调用这个函数。如果另一个线程已经占有互斥量，则进程让请求阻塞。如果互斥量没有被占有，操作系统将把占有权授予发出请求的进程，进程就可以访问资源了。

占有权仅授予那些请求该互斥量的线程，不会授予进程的所有线程。如果互斥量没有被占有，这个函数将互斥量设置成被占有，并立即返回到调用线程。如果多个线程请求同一个互斥量，具有最高优先级的线程首先被取消阻塞，并得到互斥量的占有权。如果多个阻塞线程具有相同的优先级，则使用 FIFO 顺序。

timeout 参数决定线程等待互斥量的时间长。如果在线程获得互斥量的占有权前，达到了时间限制，则返回 **ERROR_TIMEOUT**。如果指定了时间限制 **SEM_IMMEDIATE_RETURN**，函数将不阻塞线程而立即返回。线程然后可以执行其他操作，并试图在晚些时候请求互斥量。如果指定时间限制为 **SEM_INDEFINITE_WAIT**，则线程无限期等待互斥量。如果线程被一个外部事件取消阻

塞的同时，正等待互斥量，则返回 `ERROR_INTERRUPT` 给调用线程。如果试图请求互斥量的时间比允许的多，则返回 `ERROR_TOO_MANY_REQUESTS`。

事件函数

创建/打开函数

APIRET DosCreateEventSem(PSZ Name, PHEV phev, ULONG flattr, BOOL fState)

参数

Name	信号量名字的指针。
phev	事件信号量句柄的指针。
flattr	指定事件信号量属性的一套标志。
fState	信号量的初始状态。可能值有：
0 (FALSE)	信号量的初始状态为 <code>reset</code>
1 (TRUE)	信号量的初始状态为 <code>posted</code>

返回值

返回一个返回码。返回码如下：

`NO_ERROR`
`ERROR_NOT_ENOUGH_MEMORY`
`ERROR_INVALID_PARAMETER`
`ERROR_INVALID_NAME`
`ERROR_DUPLICATE_NAME`
`ERROR_TOO_MANY_HANDLES`

如果文件系统没有确认信号量名字，不包括前缀 `\SEM32\`，超过了最多的 255 个字符，则返回 `ERROR_INVALID_NAME`。如果信号量已经存在，则返回 `ERROR_DUPLICATE_NAME`。

描述

创建一个事件信号量。创建事件信号量的进程通常为控制事件或资源的进程，但不总是如此。创建信号量的进程不必在使用之前打开它。在创建信号量后，另一个进程的线程为了使用它必须打开信号量。当创建事件信号量时，标志指定了事件的初始状态，或者为已发送 (`posted`)，或者为重置 (`reset`)。如果事件处于 `reset` 状态，调用发送事件信号量函数的线程将被阻塞到可以访问信号量的进程使用发送事件信号量函数来发送事件信号量为止。如果事件初始就为发送，则调用等待事件信号量函数的线程将立即返回继续其执行。如果调用等待事件信号量函数的线程不是创建它的进程，在调用等待事件信号量函数之前，线程必须通过打开事件信号量函数打开该事件信号量。

操作系统保持一个事件信号量使用的计数。这个计数被初始化为 1。调用打开信号量函数增加该计数，调用关闭事件信号量函数减小该计数。

事件信号量可以为私有或共享。私有信号量为未命名，而且总是通过其句柄来标识。它们只可以被单个进程内的线程来使用。共享信号量可以为未命名，也可以为命名。如果信号量为命名，它可以通过名字或句柄来打开。共享信号量可以被多个进程的线程使用。

APIRET DosOpenMutex(PSZ Name, PHMTX pphmtx)

参数

Name 要打开的信号量名字指针。
pphmtx 互斥信号量句柄指针。

返回值

返回一个返回码。返回码如下：

NO_ERROR
ERROR_INVALID_HANDLE
ERROR_NOT_ENOUGH_MEMORY
ERROR_INVALID_PARAMETER
ERROR_SEM_OWNER_DIED
ERROR_INVALID_NAME
ERROR_SEM_NOT_FOUND
ERROR_TOO_MANY_OPENS

如果使用计数超过 65535，返回 ERROR_TOO_MANY_OPENS。

描述

打开一个事件信号量。这个函数被另一个进程的线程使用，让它们使用此事件信号量。事件信号量可以使用名字或句柄打开。如果使用名字，则句柄值必须为 0。如果使用句柄，名字必须为 NULL。每次调用这个函数都增加信号量的使用计数。创建信号量时，这个值被初始化为 1。当调用关闭事件信号量函数时，则减小该计数。若计数为 0，操作系统删除该信号量。使用计数不应当超过 65535。如果发生这种情况，函数将返回 ERROR_TOO_MANY_OPENS。

等待函数

APIRET DosWaitEventSem(HEV hev, ULONG TimeOut)

参数

hev 事件信号量的句柄。
TimeOut 等待事件发生的时间长。这个值可以为以下之一：
SEM_IMMEDIATE_RETURN 函数将立即返回到调用线程。

SEM_INDEFINITE_WAIT 线程将无限期等待。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_INVALID_HANDLE

ERROR_NOT_ENOUGH_MEMORY

ERROR_INTERRUPT

ERROR_TIMEOUT

如果发送事件前，达到时间限制，则返回 **ERROR_TIMEOUT**。如果在发送信号量之前等待过程被打断，则返回 **ERROR_INTERRUPT**。

描述

允许线程等待某事件信号量被发送。它可以被创建该事件信号量的进程中的任意线程调用。如果其他进程的线程通过调用打开事件信号量函数首先获得了访问信号量的权限，它们也可以调用这个函数。如果调用函数时，信号量已经被发送，这个函数将立即返回，而且线程继续运行。如果没有发送事件，则线程阻塞到发送信号量为止。如果发送事件前，达到了时间限制，则返回 **ERROR_TIMEOUT**。如果发送事件前，时间段被打断，则返回 **ERROR_INTERRUPT**。如果时间限制为 **SEM_IMMEDIATE_RETURN**，则函数立即返回到调用线程。如果时间指定为 **SEM_INDEFINITE_WAIT**，则线程无限期地等待。如果事件被发送，而且在等待线程有机会运行前被重置，则认为事件信号量在等待线程的余下时间段内被发送了。线程不必等待事件信号量再次被发送。

发送函数

APIRET DosPostEventSem(HEV hev)

参数

hev 事件信号量句柄。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_INVALID_HANDLE

ERROR_TOO_MANY_POSTS

ERROR_ALREADY_POSTED

如果调用函数时，事件信号量已被发送，则返回 **ERROR_ALREADY_POSTED**。如果超过了发送的最大数，则返回 **ERROR_TOO_MANY_POSTS**。

描述

如果事件信号量未被发送，则发送之。增加发送计数。发送计数为自从上次重置信号量后，事件信号量被发送的次数。调用事件信号量函数的所有线程将取消阻塞并恢复执行。发送事件后以及重置事件前，调用等待事件信号量函数的任何线程都将立即从调用返回到等待事件函数，而且继续执行。如果事件信号量首先重置，线程将再次阻塞。如果调用这个函数时，事件信号量已经重置，则发送计数值为 1。如果调用这个函数时，事件信号量已经发送，则函数返回 `ERROR_ALREADY_POSTED`。

创建事件的进程中的任何线程都可以发送事件信号量。如果其他进程的线程通过调用打开事件信号量函数首先获得了访问信号量的权限，它们也可以调用这个函数。

重置函数

APIRET DosResetEventSem(HEV hev, ULONG ulPostCt)

参数

hev 事件信号量句柄。

ulPostCt 接收事件信号量发送计数的指针。

返回值

返回事件信号量的发送计数。返回代码如下：

`NO_ERROR`

`ERROR_INVALID_HANDLE`

`ERROR_ALREADY_RESET`

如果已经发送事件，返回 `ERROR_ALREADY_RESET`。

描述

如果计数尚未重置，则重置发送计数为 0。属于创建事件信号量进程的任何线程都可以重置信号量。如果其他进程的线程首先调用打开事件信号量函数，它们也可以调用这个函数。发出后续调用等待事件信号量函数的所有线程都将被阻塞。当发送事件时，等待此事件信号量的所有线程都被释放，然后继续执行。

关闭函数

APIRET DosCloseEventSem(HEV hev)

参数

hev 事件信号量句柄。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_INVALID_HANDLE

ERROR_SEM_BUSY

如果线程试图在事件信号量上执行终结关闭，同时同一个进程中有其他线程正等待该事件信号量，则返回 **ERROR_SEM_BUSY**。如果事件信号量上的关闭过多，则返回 **ERROR_TOO_MANY_OPENS**。

描述

当进程不再需要访问事件信号量时，关闭事件信号量。如果进程为关闭事件信号量就终止，则由操作系统关闭它。每次调用这个函数都减小事件信号量计数。当创建事件信号量时，它被初始化为 1。每次调用打开事件信号量函数时，该计数增加。当计数值为 0 时，事件信号量从操作系统中删除。事件信号量的终结关闭减小计数值到 0，并从系统中删除事件信号量。如果线程试图对事件信号量执行终结关闭，而同时同一进程中的其他线程等待此事件信号量，则返回 **ERROR_SEM_BUSY**。

查询函数

APIRET DosQueryEventSem(HEV hev, PULONG pulPostCt)

参数

hev 事件信号量句柄。

pulPostCt 事件信号量的当前发送计数。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_INVALID_HANDLE

ERROR_INVALID_PARAMETER

如果事件信号量不存在，则返回 **ERROR_INVALID_HANDLE**。

描述

返回事件信号量的当前发送计数。发送计数为事件信号量自从上次重置后被发送的次数。如果计数值为 0，则事件信号量处于重置状态。操作系统将阻塞任何调用等待事件信号量函数的线程。创建事件信号量进程中的任何线程都可以获得事件信号量的发送计数。如果其他进程的线程

通过调用打开事件信号量函数获得了访问信号量的权限，它们也可以获得此信息。如果事件信号量不存在，函数将返回 `ERROR_INVALID_HANDLE`。

Muxwait 函数

创建/打开函数

APIRET DosCreateMuxWaitSem(PSZ Name, PHMUX phmux, ULONG ulcSemRec, PSEMRECORD pSemRec, ULONG flAttr)

参数

Name 信号量名字的指针。

phmux muxwait 信号量句柄的指针。

pSemRec 信号量记录项的计数。

FlAttr 一套定义信号量属性的标志。这些标志有：

DC_SEM_SHARED 如果设置了这个位，则共享信号量。否则，为私有信号量。

DCMW_WAIT_ANY 如果设置了这个位，当 muxwait 列表中的所有事件信号量被发送时，或者 muxwait 列表中的事件信号量被释放时，信号量清除。清除任何一个信号量后，等待 muxwait 信号量的线程可以继续执行。

DCMW_WAIT_ALL 如果设置了这个位，当 muxwait 列表中的所有事件被发送时，或者 muxwait 列表中的互斥信号量被释放时，信号量清除。清除所有信号量后，等待 muxwait 信号量的线程可以继续执行。

返回值

返回一个返回码。返回码如下：

`NO_ERROR`

`ERROR_INVALID_HANDLE`

`ERROR_INVALID_PARAMETER`

`ERROR_NOT_ENOUGH_MEMORY`

`ERROR_TOO_MANY_SEMAPHORES`

`ERROR_SEM_OWNER_DIED`

`ERROR_INVALID_NAME`

`ERROR_DUPLICATE_HANDLE`

`ERROR_DUPLICATE_NAME`

`ERROR_TOO_MANY_HANDLES`

`ERROR_WRONG_TYPE`

若列表中存在混合类型信号量，或者列表中存在 muxwait 信号量与信号量的混合类型，则返

回 `ERROR_WRONG_TYPE`。如果超过了列表上信号量最大数，则返回 `ERROR_TOO_MANY_SEMAPHORES`。如果列表中任何信号量的占有者在释放信号量前已经终结，则返回 `ERROR_SEM_OWNER_DIED`。

描述

创建多重等待 (`muxwait`) 信号量。这个函数在进程排他性同时使用若干资源时使用。多重等待信号量用于获取保护共享资源的所有互斥信号量的占有权。`muxwait` 信号量也可以用于等待一组事件信号量，让进程可以在事件发生时继续执行。

它为调用进程及其线程打开信号量。其他进程中的线程能够在其他 `muxwait` 函数中使用它之前，为了获得访问 `muxwait` 信号量的权限，它们必须首先打开 `muxwait` 信号量函数。`muxwait` 列表中的所有信号量必须在创建 `muxwait` 列表之前被创建和打开。

`muxwait` 信号量可以为私有，也可以为共享。私有信号量为未命名，因此通过其句柄来标识。共享信号量可以为未命名，也可以为命名。如果信号量为命名信号量，它可以通过名字或句柄来打开。

信号量可以包含在 `muxwait` 信号量列表中的条件如下：

_如果 `muxwait` 信号量为私有，则列表中的信号量可以共享或私有。

_如果 `muxwait` 信号量为共享，则列表中的所有信号量必须同时被共享。

_列表必须有互斥信号量或事件信号量。它不能同时包含两者，也不能包含其他 `muxwait` 信号量。

APIRET DosOpenMuxWaitSem(PSZ Name, PHMUX phmux)

参数

Name 信号量的名字。

phmuz `muxwait` 信号量的句柄。

返回值

如果成功，返回 `muxwait` 信号量的句柄。返回一个返回码。返回码如下：

`NO_ERROR`

`ERROR_INVALID_HANDLE`

`ERROR_INVALID_PARAMETER`

`ERROR_NOT_ENOUGH_MEMORY`

`ERROR_TOO_MANY_OPENS`

`ERROR_SEM_OWNER_DIED`

`ERROR_INVALID_NAME`

`ERROR_SEM_NOT_FOUND`

如果进程没有打开包含在 `muxwait` 信号量中的每个信号量，则返回 `ERROR_INVALID_HANDLE`。如果超过了允许的打开数，则返回 `ERROR_TOO_MANY_OPENS`。如果 `muxwait` 列表中的信号量占

有权没有释放信号量时已经终结, 则返回 `ERROR_SEM_OWNER_DIED`。

描述

打开一个 `mutexwait` 信号量。这个函数不是由信号量的创建者进程使用。这允许进程可以在任何 `mutexwait` 信号量函数中使用这个信号量之前获得访问 `mutexwait` 信号量的权限。属于创建 `mutexwait` 信号量进程的所有线程可以立即访问 `mutexwait` 信号量。

打开 `mutexwait` 信号量不打开包含在 `mutexwait` 列表中的所有信号量。进程在打开 `mutexwait` 信号量之前, 必须打开列表中的每个信号量, 否则, 函数将返回 `ERROR_INVALID_HANDLE`。如果打开数超过了允许的数量, 函数将返回 `ERROR_TOO_MANY_OPENS`。如果 `mutexwait` 信号量中信号量的占有者没有释放信号量而终结, 则返回 `ERROR_SEM_OWNER_DIED`。

关闭/删除函数

APIRET DosCloseMutexWait(HMux hmutex)

参数

`hmutex` `mutexwait` 信号量句柄。

返回值

返回一个返回码。返回码如下:

`NO_ERROR`

`ERROR_INVALID_HANDLE`

`ERROR_SEM_BUSY`

描述

关闭 `mutexwait` 信号量。当进程不再需要访问信号量时, 将它关闭。如果进程在关闭 `mutexwait` 信号量之前终止, 操作系统将关闭 `mutexwait` 信号量。每次调用这个函数增加信号量的计数值。当创建信号量时, 初始化计数值为 1。当计数值为 0 时, 从操作系统中删除这个信号量。计数值为 0 时, 然后发生终结关闭。如果线程试图对某个信号量执行终结关闭, 但同一个进程的其他线程正等待该信号量, 则函数返回 `ERROR_SEM_BUSY`。

APIRET DosDeleteMutexWaitSem(HMux hmutex, HSEM hsem)

参数

`hmutex` `mutexwait` 信号量句柄。

`hsem` 被删除的信号量句柄。

返回值

返回一个返回码。返回码如下:

NO_ERROR
 ERROR_INVALID_HANDLE
 ERROR_EMPTY_MUXWAIT

描述

从 muxwait 信号量中删除指定信号量。创建 muxwait 信号量的进程中任何线程都可以从列表中删除信号量。其他进程的线程也可以调用这个函数，只要它们通过调用打开 muxwait 信号量函数首先获得了访问 muxwait 信号量的权限。即使当前存在等待信号量的线程，也可以删除信号量。如果从列表中删除信号量，而且它是还没有发送或释放的唯一信号量，则等待信号量的线程被取消阻塞。如果删除的信号量为特定线程正等待的最后一个信号量，这个信号量将变成取消阻塞。如果删除的信号量为 muxwait 列表中的最后一个信号量，则所有等待这个信号量的线程都将变成取消阻塞。

查询/添加/等待函数

APIRET DosQueryMuxWaitSem(HMUX hmutex, PULONG pcSemRec, PSEMRECORD pSemRec, PULONG pflAttr)

参数

hmutex muxwait 信号量句柄。

pcSemRec 列表中可包含的信号量记录最大数的指针。

pSemRec muxwait 信号量列表中信号量记录项的指针。

pflAttr 由 muxwait 信号量创建函数传递的属性标志。有如下标志：

DC_SEM_SHARED 如果这个位被设置，信号量被共享。

DCMW_WAIT_ANY 如果设置这个位，当 muxwait 列表中的所有事件信号量被发送时，或者 muxwait 列表中的所有事件信号量被释放时，信号量清除。清除任何一个信号量后，等待 muxwait 信号量的线程可以继续执行。

DCMW_WAIT_ALL 如果设置了这个位，当 muxwait 列表中的所有事件信号量被发送时，或者 muxwait 列表中的互斥信号量被释放时，信号量清除。清除所有信号量后，等待 muxwait 信号量的线程可以继续执行。

返回值

返回一个返回码。返回码如下：

NO_ERROR
 ERROR_INVALID_HANDLE
 ERROR_INVALID_PARAMETER
 ERROR_NOT_ENOUGH_MEMORY
 ERROR_SEM_OWNER_DIED

ERROR_PARAM_TOO_SMALL

如果数组没有大到可以容纳 `mutexwait` 列表中的所有的信号量记录，则返回 `ERROR_PARAM_TOO_SMALL`。如果 `mutexwait` 列表中的互斥量占有者没有释放信号量而终止，则返回 `ERROR_SEM_OWNER_DIED`。如果 `mutexwait` 不存在，则返回 `ERROR_INVALID_HANDLE`。

描述

获取包含在 `mutexwait` 信号量中每个信号量的记录。创建 `mutexwait` 信号量进程的任何线程都可以获得这个函数提供的信息。如果其他进程的线程通过首先调用打开 `mutexwait` 信号量函数获得访问 `mutexwait` 信号量的权限，它们也可以调用这个函数。数组参数必须足够大，能够容纳这个函数提供的所有信息。如果数组不够大，函数将返回 `ERROR_PARAM_TOO_SMALL`。这个函数的记录计数参数将包含 `mutexwait` 列表中信号量记录的数量。通过读取这个参数，调用线程可以分配足够大的数组来完全容纳信号量记录。如果 `mutexwait` 列表中任何互斥信号量在释放信号量前终止，`mutexwait` 列表中所有信号量的记录都仍然被返回。函数将返回 `ERROR_SEM_OWNER_DIED`。如果 `mutexwait` 不存在，这个函数将返回 `ERROR_INVALID_HANDLE`。

APIRET DosAddMuxWaitSem(HMux hmutex, PSEMRECORD SemRec)

参数

`hmutex` `mutexwait` 信号量句柄。
`SemRec` 被添加到 `mutexwait` 列表的信号量记录的指针。

返回值

返回一个返回码。返回码如下：

`NO_ERROR`
`ERROR_INVALID_HANDLE`
`ERROR_INVALID_PARAMETER`
`ERROR_NOT_ENOUGH_MEMORY`
`ERROR_TOO_MANY_SEMAPHORES`
`ERROR_SEM_OWNER_DIED`
`ERROR_DUPLICATE_HANDLE`
`ERROR_WRONG_TYPE`

如果超过了 `mutexwait` 列表中包含的信号量最大数，则返回 `ERROR_TOO_MANY_SEMAPHORES`。如果 `mutexwait` 列表中的所有信号量都不是同一类型，则返回 `ERROR_WRONG_TYPE`。如果某线程 `w` 由于等待尚未打开此信号量的信号量，`mutexwait` 信号量无效，则返回 `ERROR_INVALID_HANDLE`。

描述

在已存在的 `mutexwait` 信号量中添加信号量。可以在线程等待 `mutexwait` 信号量时完成。创建 `mutexwait` 进程的任何线程都可以调用这个函数。其他进程的线程也可以调用这个函数，只要它们

通过调用打开 `mutexwait` 信号量函数首先获得了访问 `mutexwait` 信号量的权限。`mutexwait` 列表中的所有信号量必须为同一类型。如果在 `mutexwait` 列表中添加不同类型的信号量，函数将返回 `ERROR_WRONG_TYPE`。如果添加到 `mutexwait` 列表中的信号量超过了允许的数量（64），则函数返回 `ERROR_TOO_MANY_SEMAPHORES`。

如果函数在 `mutexwait` 列表中添加信号量，函数将检查每个线程是否等待这样的线程，即正等待着可以访问新信号量的 `mutexwait` 信号量的线程。如果有线程没有针对这个信号量调用打开函数，则函数返回 `ERROR_INVALID_HANDLE`。等待线程被取消阻塞。由那些没有让新信号量打开的进程发出的任何与 `mutexwait` 信号量有关的后续调用也返回 `ERROR_INVALID_HANDLE`。当信号量被打开时，`mutexwait` 信号量对该进程变成有效，只要 `mutexwait` 列表没有其他变化使它无效。进程必须再次调用等待 `mutexwait` 信号量函数。

APIRET DosWaitMuxWaitSem(HMux hMux, ULONG ulTimeOut, ULONG ulUser)

参数

`hMux` `mutexwait` 信号量句柄。
`ulTimeOut` 等待 `mutexwait` 信号量的时间长。
`ulUser` 被发送或释放的信号量用户域。

返回值

返回导致等待终止的信号量的用户标识符，并且返回返回码。返回码如下：

`NO_ERROR`
`ERROR_INVALID_HANDLE`
`ERROR_INVALID_PARAMETER`
`ERROR_NOT_ENOUGH_MEMORY`
`ERROR_SEM_OWNER_DIED`
`ERROR_WRONG_TYPE`
`ERROR_INTERRUPT`
`ERROR_TOO_MANY_SEM_REQUESTS`
`ERROR_MUXWAIT_EMPTY`
`ERROR_MUTEX_OWNED`
`ERROR_TIMEOUT`

如果在清除信号量之前用完了时间限制，则返回 `ERROR_TIMEOUT`。如果阻塞线程被一个外部事件取消阻塞，同时等待着 `mutexwait` 信号量，则返回 `ERROR_INTERRUPT`。如果释放信号量之前，占有者终止，则返回 `ERROR_SEM_OWNED_DIED`。如果调用线程占有 `mutexwait` 列表中的信号量，则返回 `ERROR_MUTEX_OWNED`。

描述

让线程等待一个 `mutexwait` 信号量。线程等待 `mutexwait` 列表中的信号量被发送或释放。按信号量在

列表中定义的顺序检查信号量。创建 `mutexwait` 信号量的进程的所有线程都可以调用这个函数。如果其他进程的线程通过调用打开 `mutexwait` 信号量函数首先获得访问 `mutexwait` 信号量的权限，它们也可以调用这个函数。函数将返回导致等待终止的信号量的标识符。如果调用线程必须等待 `mutexwait` 列表中的所有信号量，则函数将返回 `mutexwait` 信号量中最后一个清除信号量的标识符。时间限制参数对线程等待信号量的清除设置了一个期限。如果信号量清除前到期，则函数返回 `ERROR_TIMEOUT`。如果指定了时间限制 `SEM_IMMEDIATE_RETURN`，则函数返回而不阻塞线程。函数将继续执行，并在晚些时候调用这个函数。如果指定了时间限制 `SEM_INDEFINITE_WAIT`，线程将无限期等待并阻塞。如果外部事件取消阻塞线程，同时它正等待着 `mutexwait` 信号量，则返回返回 `ERROR_INTERRUPT`。

如果线程正等待 `mutexwait` 列表中的事件信号量，直到所有的事件信号量同时处于被发送状态，线程才会运行。如果线程正等待 `mutexwait` 列表中的所有互斥信号量被释放，直到释放所有的信号量，线程才会获得互斥量的占有权。如果线程等待 `mutexwait` 列表中的互斥信号量之一，则线程仅获得被释放的第一个信号量的占有权。`mutexwait` 列表中其他所有信号量的占有权将保持不变。如果两个线程等待具有相同优先权的相同信号量，则等待 `mutexwait` 列表中信号量的线程优先于请求单个信号量占有权的线程。也就是说前提是 `mutexwait` 列表中其他所有信号量都被释放了。如果 `mutexwait` 列表中任何互斥信号量在释放前终止，则函数返回 `ERROR_OWNER_DIED`。如果调用线程占有 `mutexwait` 列表中任何一个信号量，则函数返回 `ERROR_MUTEX_OWNED`。

临界区函数

进入/退出函数

APIRET DosEnterCritSec()

参数

无参数。

返回值

返回一个返回码。返回码如下：

`NO_ERROR`

`ERROR_INVALID_THREADID`

`ERROR_CRITSEC_OVERFLOW`

如果发生溢出，将计数设置成最大值，而且不发生任何操作，然后返回 `ERROR_CRITSEC_OVERFLOW`。如果在信号处理器或异常处理器中进入代码临界区的尝试无效，或者动态链接库程序不正确地进入临界区，则返回 `ERROR_INVALID_THREADID`。

描述

导致进程中的其他线程阻塞它们自己，并且由于线程进入临界区而放弃时间片。用一个计数

来记载进入临界区、而无相应地退出临界区的次数。当调用这个函数时增加计数，当线程退出临界区时减小计数。正常线程分配直到计数值为 0 才恢复。如果发生溢出，将计数值设置为最大值，不执行操作，而且函数返回 `ERROR_CRITSEC_OVERFLOW`。当进入信号处理器或异常处理器中代码临界区的尝试无效时，返回 `ERROR_INVALID_THREADID`。当 DLL 程序不正确发出进入临界区请求时，也返回这个返回码。

APIRET DosExitCritSec()

参数

无参数。

返回值

返回一个返回码。返回码如下：

`NO_ERROR`

`ERROR_INVALID_THREADID`

`ERROR_CRITSEC_UNDERERFLOW`

如果发生下溢，计数减小到 0 以下，无操作发生，而且返回 `ERROR_CRITSEC_UNDERFLOW`。如果退出信号处理器或异常处理器中代码临界区的尝试无效，或者动态链接库程序不正确发出退出临界区请求时，返回 `ERROR_INVALID_THREADID`。

描述

恢复正常线程，切换到进程的线程。用一个计数记录进入临界区、而没有相对应地退出临界区的次数。当线程进入临界区后，计数的值增加，调用这个函数计数减小。分派的普通线程直到计数值为 0 才被恢复。如果发生下溢，计数降到 0 以下，不发生任何操作，返回 `ERROR_CRITSEC_UNDERFLOW`。如果尝试退出信号处理器或异常处理器中代码的临界区无效，或者动态链接库程序不正确地进入和退出临界区，将返回 `ERROR_INVALID_THREADID`。

管道函数

未命名管道创建函数

APIRET DosCreatePipe(PHFILE ReadHandle, PHFILE WriteHandle, ULONG PipeSize)

参数

`ReadHandle` 读句柄的指针。

`WriteHandle` 写句柄的指针。

`PipeSize` 管道大小。

返回值

返回管道的读句柄和写句柄以及返回码。返回码如下:

NO_ERROR

ERROR_NOT_ENOUGH_MEMORY

描述

创建一个未命名管道。由这个函数返回两个管道句柄。管道大小取决于可用的内存多少。如果这个参数为 0, 则管道的缺省大小为 512 字节。管道创建者的子进程继承管道句柄。如使用共享内存时, 父进程可以传递其中一个管道句柄给子进程。一个进程可以在管道中保存数据, 其他进程可以读取这些数据。

命名管道创建函数

APIRET DosCreateNPipe(PSZ FileName, PHPIPE PipeHandle, ULONG OpenMode, ULONG PipeMode, ULONG OutBufSize, ULONG InBufSize, ULONG TimeOut)

参数

FileName	管道名字。
PipeHandle	管道句柄。
OpenMode	打开模式参数。
PipeMode	管道模式参数。
OutBufSize	输出缓冲大小。
InBufSize	输入缓冲大小。
TimeOut	DosWaitNPipe()时间耗尽缺省值。

返回值

返回一个管道句柄以及一个返回码。返回码如下:

NO_ERROR

ERROR_PATH_NOT_FOUND

ERROR_NOT_ENOUGH_MEMORY

ERROR_OUT_OF_STRUCTURES

ERROR_INVALID_PARAMETER

ERROR_PIPE_BUSY

如果客户调用打开函数时, 命名管道的所有实例都被使用, 则返回 **ERROR_PIPE_BUSY**。

描述

服务器创建一个命名管道。必须指定管道的名字, 以及访问模式、管道的类型 (字节或消息

管道), 还有输入和输出缓冲的大小。这个函数返回管道句柄。如果管道在远程计算机上打开, 客户进程必须指定服务器进程的名字, 这个服务器进程作为部分管道名字打开管道。

当服务器进程创建命名管道时, 通过指定文件直写模式 (write-through mode)、继承模式以及访问和阻塞模式来给系统定义管道。管道类型、读模式以及输入和输出缓冲的大小, 还有实例计数都必须定义。文件直写对于与远程客户进程的通信是重要的。下面描述了模式、类型和缓冲:

_当设置文件直写位, 数据一旦写入就通过网络发送。如果不设置这个位, 通过网络发送之前, 操作系统将在某些时候简短保存数据。

_访问模式指定了数据流经管道的方向。

_阻塞模式指定读和写函数是否在无数据可用时阻塞。

_继承模式决定管道句柄是否被子进程继承。

_读模式是从管道读取数据的形式。它由管道类型决定。如果管道为字节管道, 则数据按字节从管道读取。如果管道为消息管道, 则数据按消息或字节从管道读取。

_管道类型决定数据流写入管道的形式。如果管道为字节管道, 服务器和客户进程作为一致的字节流写数据。如果管道为消息管道, 则进程作为对数据阻塞的消息流写数据。每种类型都有一个系统提供的头, 它作为一个单元被写入。服务器和客户进程定义消息的大小和格式。

_输入和输出缓冲可以达到 64KB。如果管道作为消息管道来读, 则消息大小已知。服务器通过指定大小可以控制缓冲一次容纳的消息多少。

_实例计数是可以创建的命名管道的最多实例数。管道实例是一个分离的管道。它有一套特有的管道缓冲与唯一的句柄。管道实例用于区分共享相同名字的管道与具有不同名字的管道。

命名管道连接/调用/事务函数

OS2

APIRET DosConnectNPipe(HPIPE Handle)

参数

Handle 管道句柄。

返回值

返回一个返回码。返回码如下:

NO_ERROR

ERROR_INTERRUPT

ERROR_BROKEN_PIPE

ERROR_BAD_PIPE

ERROR_PIPE_NOT_CONNECTED

如果管道处于非阻塞模式, 则返回 ERROR_PIPE_NOT_CONNECTED。如果在等待客户打开管道时被打断, 则返回 ERROR_INTERRUPT。如果客户进程调用断开的命名管道函数, 则返回 ERROR_BAD_PIPE。

描述

服务器进程将命名管道设成侦听状态。它允许客户进程通过调用打开函数获得访问管道的权限。如果调用这个函数时，管道的客户终端已经打开，则函数立即返回，不起任何作用。如果调用这个函数时，客户终端关闭，其结果取决于管道是否处于阻塞模式或非阻塞模式。如果管道处于阻塞模式，函数返回前将等待客户打开管道。如果管道处于非阻塞模式，函数将立即返回，并返回 `ERROR_PIPE_NOT_CONNECTED` 返回码。管道变成侦听状态，允许客户成功调用打开函数。对某个处于非阻塞模式的管道，可以多次调用这个函数。如果管道既不打开也未关闭，首次调用这个函数将管道设成侦听状态。这个函数的后续调用测试管道状态。如果管道以前被打开，然后被客户关闭，但没有被服务器断开，函数将返回 `ERROR_BROKEN_PIPE`。如果在等待客户打开管道时，函数被打断，返回 `ERROR_INTERRUPT`。如果这个函数被客户进程所调用，则返回 `ERROR_BAD_PIPE`。

APIRET DosCallPipe(PSZ FileName, PVOID InBuffer, ULONG InBufferLen, PVOID OutBuffer, ULONG OutBufferLen, PULONG BytesOut, ULONG Timeout)

参数

FileName 被打开管道的名字。
InBuffer 被写入管道的缓冲指针。
InBufferLen 被写的字节数。
OutBuffer 返回数据的缓冲指针。
OutBufferLen 返回数据的最大字节大小。
BytesOut 系统返回实际读取的字节数的双字地址。
Timeout 等待管道实例变成可用的最长时间，以毫秒计。

返回值

返回一个返回码。返回码如下：

`NO_ERROR`
`ERROR_FILE_NOT_FOUND`
`ERROR_PIPE_BUSY`
`ERROR_BAD_FORMAT`
`ERROR_INTERRUPT`
`ERROR_BAD_PIPE`
`ERROR_PIPE_NOT_CONNECTED`
`ERROR_MORE_DATA`

如果无管道实例可用，而且时间段结束，则返回 `ERROR_INTERRUPT`。如果对一个非双向消息管道调用函数，则返回 `ERROR_BAD_FORMAT`。如果使用无效管道名，则返回 `ERROR_FILE_NOT_FOUND`。如果输出缓冲太小，不能容纳反馈消息，则返回 `ERROR_MORE_DATA`。

如果服务器进程为了将管道设成侦听状态没有调用这个函数，则返回 `ERROR_PIPE_BUSY`。

描述

综合了双向管道打开、事务和关闭函数的功能性。

APIRET DosTransactNPipe(HPIPE Handle, PVOID OutBuffer, ULONG OutBufferLen, PVOID InBuffer, ULONG InBufferLen, PULONG BytesRead)

参数

Handle 命名管道句柄。
OutBuffer 写入管道的缓冲指针。
OutBufferLen 写入管道的字节数。
InBuffer 返回数据的缓冲指针。
InBufferLen 返回数据的最大字节大小。
BytesRead 实际读取的字节数指针。

返回值

返回一个返回码。返回码如下：

`NO_ERROR`
`ERROR_PIPE_BUSY`
`ERROR_BAD_FORMAT`
`ERROR_BAD_PIPE`
`ERROR_PIPE_NOT_CONNECTED`
`ERROR_MORE_DATA`

如果对一个非双向消息管道调用此函数，则返回 `ERROR_BAD_FORMAT`。如果输入缓冲太小，不能容纳反馈消息，则返回 `ERROR_MORE_DATA`。

描述

在双向消息管道上执行一个事务。这个函数写入管道消息或从管道读取消息。如果函数用于非双向消息管道，则函数返回 `ERROR_BAD_FORMAT`。如果管道处于非阻塞模式，这个函数将整个输出缓冲写入管道。函数直到它从管道读取反馈到输入缓冲后才返回。如果输入缓冲太小，则返回 `ERROR_MORE_DATA`。如果管道中存在未读数据，则函数不成功。

命名管道断开函数

APIRET DosDisconnectNPipe(HPIPE Handle)

参数

Handle 命名管道句柄。

返回值

返回一个返回码。返回码如下：

NO_ERROR

ERROR_BROKEN_PIPE

ERROR_BAD_PIPE

如果线程阻塞于写函数，则返回 **ERROR_BROKEN_PIPE**。如果客户试图打开命名管道，则返回 **ERROR_PIPE_BUSY**。如果客户试图断开命名管道，则返回 **ERROR_BAD_PIPE**。

描述

通知客户进程已经关闭了命名管道。这个函数则被服务器进程调用。如果客户试图断开管道，函数将返回 **ERROR_BAD_PIPE**。直到服务器进程调用这个函数后，这个管道才能被另一个客户打开。通知客户的关闭后，如果服务器试图从命名管道读取，它将收到 0 字节数。如果试图写入命名管道，服务器进程将收到 **ERROR_PIPE_BROKEN**。如果客户进程试图打开命名管道，它将收到 **ERROR_PIPE_BUSY**。任何阻塞于该管道的线程都将在调用这个函数时被唤醒。阻塞于写请求的线程将收到 **ERROR_BROKEN_PIPE**。阻塞于读请求的线程将收到 0 读字节值。如果调用函数时，命名管道的客户终端打开，则强迫其关闭。客户进程将在下一个操作时收到一个错误。在这种情况下，被客户读之前，数据可能被丢弃。这个函数将使客户的句柄无效。它不会释放句柄。如果客户由于这个函数被强迫离开管道，为了释放句柄资源，客户必须调用关闭函数。

命名管道查询函数

APIRET DosQueryNPHState(HPIPE Handle, PULONG PipeHandleState)

参数

Handle 被重置命名管道的句柄。

PipeHandleState 命名管道句柄的状态。

返回值

返回与命名管道特征相关的如下信息：

 _句柄所指向的管道尾（为客户或为服务器）。

 _实例计数。

 _管道类型（或为字节类型，或为消息类型）。

 _阻塞模式（或为阻塞，或为非阻塞）。

返回返回码。返回码如下：

NO_ERROR

ERROR_BAD_PIPE

ERROR_PIPE_NOT_CONNECTED

如果对一个非双向消息管道调用这个函数, 则返回 `ERROR_BAD_FORMAT`。如果提供的输入缓冲太小, 不能容纳反馈消息, 则返回 `ERROR_MORE_DATA`。

描述

获取命名管道的相关信息。通过这个函数, 客户可以检查当前的访问模式。如果函数成功返回, 函数返回描述命名管道特征的信息。

实例计数的值以及管道类型不能更改。它们与创建管道时指定的一样。阻塞模式和读模式的返回信息可以来自不同的源。客户调用打开函数时, 如果句柄是针对命名管道客户终端, 则由系统设置阻塞模式和读模式为阻塞和字节读。它们仍然能被这个函数重置。如果是针对管道的服务器终端, 则创建命名管道时, 设置阻塞模式和读模式。它们也可以被这个函数重置。如果另一个线程阻塞于同一个管道的读或写操作, 管道不能改为无等待模式。

APIRET DosQueryNPipeInfo(HPIPE Handle, ULONG InfoLevel, PVOID InfoBuf, ULONG InfoBufSize)

参数

Handle 命名管道句柄。
InfoLevel 需求的管道数据。
InfoBuf 管道信息数据结构。
InfoBufSize 管道数据缓冲大小。

返回值

返回信息数据结构, 它包含与命名管道当前状态相关的信息。它返回 1 级和 2 级信息。1 级信息如下:

_输入和输出缓冲大小。
 _允许的命名管道实例最大数。
 _当前命名管道实例数。
 _管道的 ASCII 名字。

2 级信息包含一个针对每个客户管道进程的特有 2 字节标识符。返回码如下:

`NO_ERROR`
`ERROR_BUFFER_OVERFLOW`
`ERROR_INVALID_LEVEL`
`ERROR_BAD_PIPE`

描述

获取命名管道的相关信息。这个函数的信息比 `DosQueryNPHState` 函数更详细。这个函数返回一个 `PIPEINFO` 数据结构。它包含如下信息:

_输入和输出缓冲的大小。

_允许的命名管道实例最大数。

_命名管道实例的当前数。

_管道的 ASCIIZ 名字。

APIRET DosQueryNPipeSemState(HSEM SemHandle, PPIPESEMSTATE InfoBuf, ULONG InfoBufLen)

参数

SemHandle 先前附加到一个或多个命名管道上的共享事件或 muxwait 信号量的句柄。

InfoBuf 缓冲的指针, 这个缓冲包含附加于信号量的每个命名管道的记录。在每条记录中包含如下域:

fStatus 告知命名管道状态的编码值。

fFlag 表明命名管道额外信息的位域。

usKey 当调用设置命名管道信号量函数时, **KeyHandle** 的指定值。

UsAvail 如果 **fStatus** 的值为 1, 这个域将包含命名管道中可读的数据字节数。如果 **fStatus** 的值为 2, 这个域将包含管道中可写的字节数。

InfoBufLen 包含附加到信号量上的每个命名管道记录的缓冲大小, 以字节为单位。

返回值

返回一个返回码。返回码如下:

NO_ERROR

ERROR_INVALID_PARAMETER

ERROR_BUFFER_OVERFLOW

描述

返回附加于共享事件或 muxwait 信号量上的局部命名管道状态相关的信息。管道的状态可以是关闭, 或者允许执行阻塞模式的输入和输出。如果由多个进程在不同的命名管道上附加同一个信号量, 则不能返回调用者不能访问的命名管道相关信息。如果进程希望缓冲中的数据只引用它自己的命名管道, 进程必须是一个私有事件信号量。

APIRET DosPeekNPipe(HPIPE Handle, PVOID Buffer, ULONG BufferLen, PULONG BytesRead, PAVAILDATA BytesAvail, PULONG PipeState)

参数

Handle 被检查的命名管道句柄。

Buffer 输出缓冲的指针。

BufferLen 读入的字节数。

BytesRead 实际读入的字节数指针。

BytesAvail 缓冲的指针, 系统返回这个缓冲上可用的字节数。

PipeState 表示命名管道状态的值指针。这些值有：

- 1 (NP_STATE_DISCONNECTED) 断开连接
- 2 (NP_STATE_LISTENING) 侦听
- 3 (NP_STATE_CONNECTED) 连接
- 4 (NP_STATE_CLOSING) 关闭

返回值

返回命名管道相关信息、管道状态以及一个返回码。返回码如下：

NO_ERROR
 ERROR_PIPE_BUSY
 ERROR_BAD_PIPE
 ERROR_PIPE_NOT_CONNECTED

如果不能立即访问管道，则返回 ERROR_PIPE_BUSY。

描述

检查命名管道中的数据，而不从管道中删除它们。这个函数返回管道状态的相关信息。这个函数从不阻塞。不能立即访问命名管道，函数将返回 ERROR_PIPE_BUSY。函数只返回管道中的当前内容。如果管道中的消息被检查，只返回部分消息。

命名管道设置函数

APIRET DosSetNPipeSem(HPIPE Handle, HSEM SemHandle, ULONG KeyHandle)

参数

- Handle 信号量所附加的命名管道句柄。
- SemHandle 当管道有数据可读或可写时，被发送的事件信号量或 muxwait 信号量句柄。
- KeyHandle 区别事件到达于附加在同一信号量上的不同命名管道句柄的键值。

返回值

返回一个返回码。返回码如下：

NO_ERROR
 ERROR_INVALID_PARAMETER
 ERROR_INVALID_FUNCTION
 ERROR_INVALID_HANDLE
 ERROR_SEM_NOT_FOUND
 ERROR_BAD_PIPE
 ERROR_PIPE_NOT_CONNECTED

如果试图附加信号量到一个远程管道，则返回 ERROR_INVALID_FUNCTION。

描述

在局部命名管道上附加一个共享事件信号量。如果试图附加信号量到一个远程管道，则函数返回 `ERROR_INVALID_FUNCTION`。

APIRET DosSetNPHState(HPIPE Handle, ULONG PipeHandleState)

参数

Handle 被重置的命名管道句柄。

PipeHandleState 命名管道句柄状态。

返回值

返回一个返回码。返回码如下：

`NO_ERROR`

`ERROR_BAD_PIPE`

`ERROR_INVALID_PARAMETER`

`ERROR_PIPE_BUSY`

`ERROR_PIPE_NOT_CONNECTED`

描述

重置命名管道的阻塞模式和读模式。必须指定阻塞和读模式。如果命名管道为字节管道，读模式不能更改。如果另一个线程在同一个管道终端正阻塞于一个 I/O 请求，则阻塞模式不能更改为非阻塞模式。

命名管道等待函数

APIRET DosWaitNPipe(PSZ FileName, ULONG TimeOut)

参数

FileName 被打开管道的 ASCIIZ 名字的指针。

TimeOut 等待命名管道实例变成可用的最长时间。

返回值

返回一个返回码。返回码如下：

`NO_ERROR`

`ERROR_FILE_NOT_FOUND`

`ERROR_INTERRUPT`

`ERROR_PIPE_BUSY`

描述

当管道的所有实例繁忙时，允许客户进程等待命名管道实例变成可用。这个参数决定等待命名管道实例变成可用的最长时间。如果命名管道实例变成可用前，到达了时限，则函数返回 **ERROR_PIPE_BUSY**。如果时限值为 0，则系统使用创建命名管道时指定的缺省时间耗尽值。如果时限值为 _1，则发生无限期等待。如果多个客户等待命名管道的某个实例，则系统把管道实例交给具有最高优先权线程的客户进程。如果所有的线程优先权相同，则等待最长时间的线程将得到下一个可用的管道实例。

附录 F 线程和同步类

(POSIX, Win32 以及 OS/2)

POSIX 示例类

```
ct_thread class header.

#include <pthread.h>
typedef void *(*FunctionPtr) (void*);

class ct_thread{
private:
    pthread_t ThreadId;
    pthread_attr_t *Attr;
public:
    ct_thread(void);
    ct_thread(pthread_attr_t *Attribute);
    ~ct_thread(void);
    void begin(FunctionPtr PFN,void *X);
    void wait(void);
    pthread_t threadHandle(void);
    pthread_t threadId(void);
};
ct_thread class defintion.

#include "cthread.h"
```



```
ct_thread::ct_thread(void)
{
    Attr = NULL;
}

ct_thread::ct_thread(pthread_attr_t *Attribute)
{
    Attr = Attribute;
}

ct_thread::~~ct_thread(void)
{
}

void ct_thread::begin(FunctionPtr PFN, void *X)
{
    pthread_create(&ThreadId, Attr, PFN, X);
}

void ct_thread::wait(void)
{
    pthread_join(ThreadId, NULL);
}

pthread_t ct_thread::threadHandle(void)
{
    return(ThreadId);
}

pthread_t ct_thread::threadId(void)
{
    return(pthread_self());
}
```

mutex class header.

```
#include <eclasses.h>
#include <pthread.h>

class mutex{
protected:
    pthread_mutex_t Mutex;
    general_exception SemException;
```

```

public:
    mutex(void);
    ~mutex(void);
    void lock(void);
    void unlock(void);
};

```

mutex class definition.

```

#include <String.h>
#include <ctmutex.h>

```

```

mutex::mutex(void)
{
    if(pthread_mutex_init(&Mutex,NULL)){
        SemException.message("Could Not Create Mutex");
        throw SemException;
    }
}

mutex::~~mutex(void)
{
    if(pthread_mutex_destroy(&Mutex)){
        SemException.message("Could Not Destroy Mutex");
        throw SemException;
    }
}

void mutex::lock(void)
{
    pthread_mutex_lock(&Mutex);
}

void mutex::unlock(void)
{
    pthread_mutex_unlock(&Mutex);
}

```

named_pipe class header.

```

#include <string.h>
#include <fstream.h>

```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <unistd.h>
#include <vector.h>
#include <algo.h>
```

```
template <class T> class named_pipe{
private:
    char PipeName[20];
    unsigned long PipeHandle;
    fstream NamedPipe;
public:
    named_pipe(char *PName, int Mode);
    ~named_pipe(void);
    void operator<<(const T &X);
    void operator<<(const vector<T> &X);
    void operator>>(T &X);
};
```

named_pipe class definition.

```
template <class T> named_pipe<T>::named_pipe(char *PName,
                                              int Mode)
{
    unlink(PName);
    mknod(PName, S_IFIFO, 0);
    chmod(PName, 0660);
    if((PipeHandle = open(PName, Mode)) == -1)
    {
        cerr << "could not open fifo with open " << endl;
    }
    while(PipeHandle == -1)
    {
        sleep(1);
        PipeHandle = open(PName, Mode);
    }
    NamedPipe.attach(PipeHandle);
    if(!NamedPipe.good()){
        cerr << "could not attach handle" << endl;
    }
}
```

```

template <class T> named_pipe<T>::~named_pipe(void)
{
    NamedPipe.close();
}

template <class T> void named_pipe<T>::operator<<(const T &X)
{
    NamedPipe << X << " ";
}

template <class T> void named_pipe<T>::operator>>(T &X)
{
    NamedPipe >> X;
}

template <class T> void named_pipe<T>::operator<<
    (const vector<T> &X)
{
    ostream_iterator<T> Out(NamedPipe, " ");
    copy(X.begin(), X.end(), Out);
}

```

OS/2 示例类

```

ct_thread class header.

typedef void (* FunctionPtr) (void*);
class ct_thread{
private:
    unsigned long ThreadId;
    int StackSize;
public:
    ct_thread(int SSize = 4096);
    ~ct_thread(void);
    void begin(FunctionPtr PFN, void *X);
    void wait(void);
    unsigned long threadId(void);
};

```

ct_thread class definition.

```
#define INCL_DOSPROCESS
#include <os2.h>
#include "ctthread.h"
#include <stddef.h>
#include <process.h>
```

```
ct_thread::ct_thread(int SSize)
{
    StackSize = SSize;
    ThreadId = 0;
}
```

```
ct_thread::~ct_thread(void)
{
    DosKillThread(ThreadId);
}
```

```
void ct_thread::begin(FunctionPtr PFN, void *X)
{
    ThreadId = _beginthread(PFN, StackSize, X);
}
```

```
void ct_thread::wait(void)
{
    DosWaitThread(&ThreadId, DCWW_WAIT);
}
```

```
unsigned long ct_thread::threadId(void)
{
    return(ThreadId);
}
```

event_mutex class header.

```
#include "eclasses.h"
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
```

```

#include <os2.h>

class event_mutex{
protected:
    HEV EventMutex;
    char MutexName[81];
    int InitallySet;
    general_exception EventException;
    unsigned long EventWait;
public:
    event_mutex(char *MName, int Initial = 0,
                unsigned long Dur = SEM_INDEFINITE_WAIT);
    void postEvent(void);
    void waitEvent(void);
    ~event_mutex(void);
};

event_mutex class definition.

#include "ctevent.h"

event_mutex::event_mutex(char *MName, int Initial, unsigned long Dur)
{
    if(DosCreateEventSem(MName,&EventMutex,0,Initial)){
        EventException.message("Could Not Create Event
                               Semaphore");
        throw EventException;
    }
    EventWait = Dur;
}

void event_mutex::postEvent(void)
{
    DosPostEventSem(EventMutex);
}

void event_mutex::waitEvent(void)
{
    DosWaitEventSem(EventMutex,EventWait);
}

```

```
event_mutex::~event_mutex(void)
{
    if(DosCloseEventSem(EventMutex)){
        EventException.message("Could Not Close Mutex");
        throw EventException;
    }
}
```

multiple_event_mutex class header.

```
class multiple_event_mutex{
protected:
    char MutexName[81];
    SEMRECORD *ConditionList;
    int EventNumber;
    long int WaitSelection;
    HEV EventMutex;
    HMUX MultipleEventMutex;
    general_exception EventException;
    int Duration;
public:
    multiple_event_mutex(char *MName, int ENum, long int
                        WaitType, int Initial);
    multiple_event_mutex(char *MName);
    ~multiple_event_mutex(void);
    int duration(void);
    void duration(int X);
    void postEvent(int X);
    void waitEvents(void);
};
```

multiple_event_mutex class definition.

```
#include "ctevent.h"
```

```
multiple_event_mutex::multiple_event_mutex(char *MName, int ENum, long int
                        WaitType, int Initial)
{
    int N;
    EventNumber = ENum;
    ConditionList = new SEMRECORD[ENum];
    for(N = 0; N < ENum; N++)
```

```

    {
        if(DosCreateEventSem(NULL, &EventMutex, 0, Initial)){
            EventException.message("Could Not Create Event Semaphore");
            throw EventException;
        }
        ConditionList[N].hsemCur = &EventMutex;
        ConditionList[N].ulUser = N;
    }
    if(DosCreateMuxWaitSem(MName, &MultipleEventMutex,
        sizeof(ConditionList),
        ConditionList, WaitType)){
        EventException.message("Could Not Create Multiple Event Semaphore");
        throw EventException;
    }
}

multiple_event_mutex::multiple_event_mutex(char *MName)
{
    if(DosOpenMuxWaitSem(MName, &MultipleEventMutex)){
        EventException.message("Could Not Open Event Semaphore");
        throw EventException;
    }
}

multiple_event_mutex::~multiple_event_mutex(void)
{
    if(DosCloseMuxWaitSem(MultipleEventMutex)){
        EventException.message("Could Not Close Event Semaphore");
        throw EventException;
    }
    delete ConditionList;
}

int multiple_event_mutex::duration(void)
{
    return(Duration);
}

void multiple_event_mutex::duration(int X)
{
    Duration = X;
}

void multiple_event_mutex::postEvent(int X)
{

```



```
        EventMutex = (HEV) ConditionList[X].hsemCur;
        DosPostEventSem(EventMutex);
    }

    void multiple_event_mutex::waitEvents(void)
    {
        unsigned long User;
        DosWaitMuxWaitSem(MultipleEventMutex, SEM_INDEFINITE_WAIT, &User);
    }
```

mutex class header.

```
#include "eclasses.h"
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>
#include <cstring.h>

class mutex{
protected:
    unsigned long Duration;
    HMTX Mutex;
    general_exception SemException;
public:
    mutex(void);
    ~mutex(void);
    virtual void lock(void);
    virtual void unlock(void);
};
```

mutex class definition.

```
#include <string.h>
#include "ctmutex.h"
#include <iostream.h>

mutex::mutex(void)
{
    if(DosCreateMutexSem(NULL, &Mutex, 0, 0)){
        SemException.message("Could Not Create Mutex");
        throw SemException;
    }
```

```

    Duration = SEM_INDEFINITE_WAIT;
}

mutex::~mutex(void)
{
    if(DosCloseMutexSem(Mutex)){
        SemException.message("Could Not Destroy Mutex");
        throw SemException;
    }
}

void mutex::lock(void)
{
    DosRequestMutexSem(Mutex, Duration);
}

void mutex::unlock(void)
{
    DosReleaseMutexSem(Mutex);
}

named_mutex class header.

class named_mutex : public mutex{
protected:
    char MutexName[81];
    int InitiallyOwned;
public:
    named_mutex(void);
    named_mutex(char *MName, int Owned = 0);
    named_mutex(string MName);
    unsigned long lockDuration(void);
    void lockDuration(unsigned long Dur);
};

named_mutex class definition.

named_mutex::named_mutex(void)
{

```

```
        if(DosCreateMutexSem(NULL, &Mutex, 0, 0)){
            SemException.message("Could Not Create Mutex");
            throw SemException;
        }
        Duration = SEM_INDEFINITE_WAIT;
    }

    named_mutex::named_mutex(char *MName, int Owned)
    {
        strcpy(MutexName, MName);
        InitiallyOwned = Owned;
        if(DosCreateMutexSem(MName, &Mutex, 0, InitiallyOwned)){
            SemException.message("Could Not Create Mutex");
            throw SemException;
        }
        Duration = SEM_INDEFINITE_WAIT;
    }

    named_mutex::named_mutex(string MName)
    {
        unsigned long RC;
        strcpy(MutexName, MName.c_str());
        InitiallyOwned = 0;
        Mutex = (HMTX) NULL;
        RC = DosOpenMutexSem(MutexName, &Mutex);
        if(RC){
            cout << RC << endl;
            SemException.message("Could Not open Mutex");
            throw SemException;
        }
        Duration = SEM_INDEFINITE_WAIT;
    }

    unsigned long named_mutex::lockDuration(void)
    {
        return(Duration);
    }

    void named_mutex::lockDuration(unsigned long Dur)
    {
        Duration = Dur;
    }
```

lqueue class header.

```
#include <deque.h>
#include "ctmutex.h"

template <class T> class lqueue : virtual private named_mutex, virtual private
    event_mutex{
protected:
    deque<T> SafeQueue;
public:
    lqueue(char *MName,int Own,char *EName,int Initial, unsigned long Dur);
    inline void insert(T X);
    inline T remove(void);
    inline T front(void);
    inline T back(void);
    inline unsigned int empty(void);
    inline unsigned int size(void);
    inline void erase(void);
    inline void reversed(void);
    void wait(void);
    void broadCast(void);
};
```

lqueue class definition.

```
#include "ctqueue.h"
#include "ctevent.h"

template <class T> lqueue<T>::lqueue(char *MName,int Own, char *EName,
    int Initial,unsigned long Dur) :
    named_mutex(MName,Own),
    event_mutex(EName,Initial,Dur)
{
}

template <class T> void lqueue<T>::insert(T X)
{
    lock();
    SafeQueue.push_back(X);
    unlock();
}

template <class T> T lqueue<T>::remove(void)
```

```
{

    T Temp;
    lock();
    Temp = SafeQueue.front();
    SafeQueue.pop_front();
    unlock();
    return(Temp);
}

template <class T> void lqueue<T>::reversed(void)
{
    lock();
    reverse(SafeQueue.begin(), SafeQueue.end());
    unlock();
}

template <class T> T lqueue<T>::front(void)
{
    return(SafeQueue.front());
}

template <class T> T lqueue<T>::back(void)
{
    return(SafeQueue.back());
}

template <class T> unsigned int lqueue<T>::size(void)
{
    return(SafeQueue.size());
}

template <class T> unsigned int lqueue<T>::empty(void)
{
    return(SafeQueue.empty());
}

template <class T> void lqueue<T>::erase(void)
{
    lock();
    SafeQueue.erase(SafeQueue.begin(), SafeQueue.end());
    unlock();
}
```

```
template <class T> void lqueue<T>::wait(void)
{
    waitEvent();
}
```

```
template <class T> void lqueue<T>::broadcast(void)
{
    postEvent();
}
```

mt_set class header.

```
#include "set.h"
#include "algo.h"
#include "ctmutex.h"
```

```
template <class T> class mt_set : virtual private mutex{
private:
    set<T,less<T> > S;
    set<T,less<T> > M;
public:
    mt_set(void);
    mt_set(set<T,less<T> > X);
    set<T,less<T> > setUnion(set<T,less<T> > X);
    set<T,less<T> > intersection(set<T,less<T> > X);
    int membership(set<T,less<T> > X);
    set<T,less<T> > difference(set<T,less<T> > X);
};
```

mt_set class definition.

```
template <class T> mt_set<T>::mt_set(void)
{
}
```

```
template <class T> mt_set<T>::mt_set(set<T, less<T> > X)
{
    S = X;
```

```
    }

    template <class T> set<T, less<T> > mt_set<T>::intersection(set<T, less<T> >
X)
    {
        set<T, less<T> > Temp;
        less<T> Order;
        lock();
        set_intersection(S.begin(), S.end(), X.begin(), X.end(),
            inserter(Temp, Temp.begin()), Order);
        unlock();
        return(Temp);
    }

    template <class T> set<T, less<T> > mt_set<T>::setUnion(set<T, less<T> > X)
    {
        less<T> Order;
        set<T, less<T> > Temp;
        lock();
        set_union(S.begin(), S.end(), X.begin(), X.end(),
            inserter(Temp, Temp.begin()), Order);
        unlock();
        return(Temp);
    }

    template <class T> int mt_set<T>::membership(set<T, less<T> >X)
    {
        return(includes(S.begin(), S.end(), X.begin(), X.end()));
    }

    template <class T> set<T, less<T> > mt_set<T>::difference(set<T, less<T> > X)
    {
        set<T, less<T> > Temp;
        less<T> Order;
        lock();
        set_difference(S.begin(), S.end(), X.begin(), X.end(),
            inserter(Temp, Temp.begin()), Order);
        unlock();
        return(Temp);
    }
}
```

npstream class header.

```

#define INCL_DOSNMPIPES
#define INCL_DOSQUEUES
#include <string.h>
#include <fstream.h>
#include<os2.h>
#include<vector.h>
#include<algo.h>

template <class T> class npstream{
private:
    char PipeName[20];
    unsigned long PipeHandle;
    long int PipeMode;
    long int OpenMode;
    long int OutputBufSize;
    long int InputBufSize;
    long int TimeOut;
    unsigned long Result;
    fstream NamedPipe;
public:
    npstream(char *PName,long PMode,long OMode, long TOut = 1000);
    npstream(char *PName);
    ~npstream(void);
    void operator<<(T &X);
    void operator<<(vector<T> &X);
    void operator>>(T &X);
    long int pipeMode(void);
    long int openMode(void);
    long int outBufSize(void);
    long int inBufSize(void);
    long int timeOut(void);
    unsigned long result(void);
};

npstream class definition.
template <class T> npstream<T>::npstream(char *PName,
                                           long PMode,
                                           long OMode,
                                           long TOut = 1000)
{
    strcpy(PipeName,PName);
    PipeMode = PMode;

```



```
        OpenMode = OMode;
        OutputBufSize = (sizeof(T) * 1000);
        InputBufSize = (sizeof(T) * 1000);
        TimeOut = TOut;
        Result = DosCreateNPipe(PipeName,
                                &PipeHandle,
                                OpenMode,
                                PipeMode,
                                OutputBufSize,
                                InputBufSize, TimeOut);
        DosConnectNPipe(PipeHandle);
        NamedPipe.attach(PipeHandle);

    }

template <class T> npstream<T>::npstream(char *PName)
{
    DosOpen(PName, &PipeHandle, &Result, 0,
            FILE_NORMAL, FILE_OPEN,
            OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
            (PEAOP) NULL);
    NamedPipe.attach(PipeHandle);
}

template <class T> npstream<T>::~npstream(void)
{
    NamedPipe.close();
}

template <class T> void npstream<T>::operator<<(T &X)
{
    NamedPipe << X << " ";
}

template <class T> void npstream<T>::operator>>(T &X)
{
    NamedPipe >> X;
}

template <class T> void npstream<T>::operator<<(vector<T> &X)
```

```

{

    ostream_iterator<T> Out(NamedPipe, " ");
    copy(X.begin(), X.end(), Out);

}

template <class T> long int npstream<T>::pipeMode(void)
{
    return(PipeMode);
}

template <class T> long int npstream<T>::openMode(void)
{
    return(OpenMode);
}

template <class T> long int npstream<T>::outBufSize(void)
{
    return(OutputBufSize);
}

template <class T> long int npstream<T>::inBufSize(void)
{
    return(InputBufSize);
}

template <class T> long int npstream<T>::timeOut(void)
{
    return(TimeOut);
}

template <class T> unsigned long npstream<T>::result(void)
{
    return(Result);
}

set_server class definition.

#include "cthread.h"
#include <set.h>
#include "mtset.cpp"
#include <cstring.h>
#include <process.h>

```

```
#define INCL_DOSPROCESS
#include <os2.h>

template <class T> struct server_argument{
    set<T,less<T> > A;
    set<T,less<T> > B;
    set<T,less<T> > Result;
};

template <class T> class set_server : virtual private mutex{
protected:
    server_argument<T> Argument;
public:
    set_server(void);
    friend void intersection(void *X);
    friend void setUnion(void *X);
    friend void difference(void *X);
    friend void membership(void *X);
    set<T,less<T> > intersect(set<T,less<T> > X, set<T,less<T> > Y);
    set<T,less<T> > setUnion(set<T,less<T> > &X, set<T,less<T> > &Y);
    set<T,less<T> > difference(set<T,less<T> > X, set<T,less<T> > Y);
    int membership(set<T,less<T> > X, set<T,less<T> > Y);
};

set_server class definition.

template <class T> set_server<T>::set_server(void)
{
}

template <class T> set<T,less<T> > set_server<T>::intersect(set<T,less<T>
> X, set<T,less<T> > Y)
{
    lock();
    ct_thread Thread(8192);
    Argument.A = X;
    Argument.B = Y;
    Thread.begin(intersection,this);
    Thread.wait();
    unlock();
    return(Argument.Result);
}
```

```

template <class T> set<T,less<T> > set_server<T>::setUnion(set<T,less<T> >
&X, set<T,less<T> > &Y)
{
    lock();
    ct_thread Thread(8192);
    Argument.A = X;
    Argument.B = Y;
    Thread.begin(::setUnion,this);
    Thread.wait();
    unlock();
    return(Argument.Result);
}

template <class T> int set_server<T>::membership(set<T,
less<T> > X, set<T,less<T> > Y)
{
    lock();
    ct_thread Thread(8192);
    Argument.A = X;
    Argument.B = Y;
    Thread.begin(::membership,this);
    Thread.wait();
    unlock();
    if(Argument.Result.size()){
        return(1);
    }
    else{
        return(0);
    }
}

template <class T> set<T,less<T> > set_server<T>::difference(set<T,less<T>
> X, set<T,less<T> >Y)
{
    lock();
    ct_thread Thread(8192);
    Argument.A = X;
    Argument.B = Y;
    Thread.begin(::difference,this);
    Thread.wait();
    unlock();
    return(Argument.Result);
}

```

```
void intersection(void *X)
{
    set_server<string> *Server;
    Server = static_cast<set_server<string> *> (X);
    mt_set<string> SafeSet(Server->Argument.A);
    Server->Argument.Result = SafeSet.intersection(Server->Argument.B);
}

void setUnion(void *X)
{
    set_server<string> *Server;
    Server = static_cast<set_server<string> *> (X);
    mt_set<string> SafeSet(Server->Argument.A);
    Server->Argument.Result = SafeSet.setUnion(Server->Argument.B);
}

void difference(void *X)
{
    set_server<string> *Server;
    Server = static_cast<set_server<string> *> (X);
    mt_set<string> SafeSet(Server->Argument.A);
    Server->Argument.Result = SafeSet.difference(Server->Argument.B);
}

void membership(void *X)
{
    set_server<string> *Server;
    Server = static_cast<set_server<string> *> (X);
    mt_set<string> SafeSet(Server->Argument.A);
    if(SafeSet.membership(Server->Argument.B)){
        Server->Argument.Result.insert("one");
    }
}
```

Win32 示例类

ct_thread class declaration.

```

typedef void (* FunctionPtr) (void*);
class ct_thread{
private:
    DWORD ThreadId;
    HANDLE ThreadHandle;
    DWORD StackSize, ExitCode;

public:
    ct_thread(DWORD SSize = 4096);
    ~ct_thread(void);
    void begin(FunctionPtr PFN, void *X);
    void wait(void);
    unsigned long threadId(void);
};

```

ct_thread class declaration.

```

#include <windows.h>
#include <process.h>
#include "ctthread.h"
#include <stddef.h>

```

```

ct_thread::ct_thread(DWORD SSize)
{
    StackSize = SSize;
    ThreadId = 0;
}

ct_thread::~ct_thread(void)
{
    TerminateThread(ThreadHandle, ExitCode);
}

void ct_thread::begin(FunctionPtr PFN, void *X)
{
    ThreadHandle = CreateThread(NULL, StackSize,
(LPTHREAD_START_ROUTINE) PFN, (X,0,&ThreadId);
}

void ct_thread::wait(void)
{
    WaitForSingleObject(ThreadHandle, INFINITE);
}

```

```
}
```

```
unsigned long ct_thread::threadId(void)
```

```
{
```

```
    return(ThreadId);
```

```
}
```

event_mutex class declaration.

```
#include "eclasses.h"
```

```
#include <windows.h>
```

```
class event_mutex{
```

```
protected:
```

```
    HANDLE EventMutex;
```

```
    char MutexName[81];
```

```
    int InitallySet;
```

```
    general_exception EventException;
```

```
    DWORD EventWait;
```

```
public:
```

```
    event_mutex(char *MName, BOOL Initial = 0, unsigned long Dur = INFINITE);
```

```
    void postEvent(void);
```

```
    void waitEvent(void);
```

```
    ~event_mutex(void);
```

```
};
```

event_mutex class definition.

```
#include "ctevent.h"
```

```
#include <windows.h>
```

```
event_mutex::event_mutex(char *MName, BOOL Initial, unsigned long Dur)
```

```
{
```

```
    strcpy(MutexName,MName);
```

```
    EventMutex = CreateEvent(NULL,FALSE,Initial,MutexName);
```

```
    if(!EventMutex){
```

```

        EventException.message("Could Not Create Event Semaphore");
        throw EventException;
    }
    EventWait = Dur;
}

void event_mutex::postEvent(void)
{
    SetEvent(EventMutex);
}

void event_mutex::waitEvent(void)
{
    WaitForSingleObject(EventMutex, EventWait);
}

event_mutex::~event_mutex(void)
{
    if(!CloseHandle(EventMutex)){
        EventException.message("Could Not Close Mutex");
        throw EventException;
    }
}

mutex class declaration.

#include "eclases.h"
#include <windows.h>
#include <cstring.h>

class mutex{
protected:
    DWORD Duration;
    HANDLE Mutex;
    general_exception SemException;
public:
    mutex(void);
    ~mutex(void);
    virtual void lock(void);
    virtual void unlock(void);
};

```


mutex class definition.

```
#include <cstring.h>
#include <windows.h>
#include <process.h>
#include <string.h>
#include "ctmutex.h"
#include <iostream.h>
```

mutex::mutex(void)

```
{
    Mutex = CreateMutex(NULL,0,NULL);
    if(!Mutex){
        SemException.message("Could Not Create Mutex");
        throw SemException;
    }
    Duration = INFINITE;
}
```

mutex::~~mutex(void)

```
{
    if(!CloseHandle(Mutex)){
        SemException.message("Could Not Destroy Mutex");
        throw SemException;
    }
}
```

void mutex::lock(void)

```
{
    WaitForSingleObject (Mutex,Duration);
}
```

void mutex::unlock(void)

```
{
    ReleaseMutex(Mutex);
}
```

named_mutex class declaration.

```
class named_mutex : public mutex{
```

```

protected:
    char MutexName[81];
    BOOL InitiallyOwned;
public:
    named_mutex(void);
    named_mutex(char *MName, BOOL Owned = 0);
    named_mutex(string MName);
    DWORD lockDuration(void);
    void lockDuration(DWORD Dur);
};

named_mutex class definition.

named_mutex::named_mutex(void)
{
    Mutex = CreateMutex(NULL, 0, NULL);
    if(!Mutex){
        SemException.message("Could Not Create Mutex");
        throw SemException;
    }
    Duration = INFINITE;
}

named_mutex::named_mutex(char *MName, BOOL Owned)
{
    strcpy(MutexName, MName);
    InitiallyOwned = Owned;
    Mutex = CreateMutex(NULL, InitiallyOwned, MutexName);
    if(!Mutex){
        SemException.message("Could Not Create Mutex");
        throw SemException;
    }
    Duration = INFINITE;
}

named_mutex::named_mutex(string MName)
{
    strcpy(MutexName, MName.c_str());
    InitiallyOwned = 0;
    Mutex = OpenMutex(SYNCHRONIZE, TRUE, MutexName);
    if(!Mutex){
        SemException.message("Could Not open Mutex");
    }
}

```

```
        throw SemException;
    }
    Duration = INFINITE;
}
DWORD named_mutex::lockDuration(void)
{
    return(Duration);
}

void named_mutex::lockDuration(DWORD Dur)
{
    Duration = Dur;
}
```

lqueue class declaration.

```
#include <deque.h>
#include "ctmutex.h"

template <class T> class lqueue : virtual private named_mutex,
virtual private event_mutex{
protected:
    deque<T> SafeQueue;
public:
    lqueue(char *MName,int Own,char *FName,int Initial,
        unsigned long Dur);
    inline void insert(T X);
    inline T remove(void);
    inline T front(void);
    inline T back(void);
    inline unsigned int empty(void);
    inline unsigned int size(void);
    inline void erase(void);
    inline void reversed(void);
    void wait(void);
    void broadCast(void);

};
```

lqueue class definition.

```
#include <cstring.h>
#include <windows.h>
#include "ctqueue.h"
#include "ctevent.h"
```

```
template <class T> lqueue<T>::lqueue(char *MName,int Own,char*EName,int
                                     Initial, unsigned long Dur):named_mutex
                                     (MName,Own), event_mutex(EName,Initial,Dur)
{
}

template <class T> void lqueue<T>::insert(T X)
{
    lock();
    SafeQueue.push_back(X);
    unlock();
}

template <class T> T lqueue<T>::remove(void)
{
    T Temp;
    lock();
    Temp = SafeQueue.front();
    SafeQueue.pop_front();
    unlock();
    return(Temp);
}

template <class T> void lqueue<T>::reversed(void)
{
    lock();
    reverse(SafeQueue.begin(),SafeQueue.end());
    unlock();
}

template <class T> T lqueue<T>::front(void)
{
    return(SafeQueue.front());
}

template <class T> T lqueue<T>::back(void)
{
    return(SafeQueue.back());
}
```

```
template <class T> unsigned int lqueue<T>::size(void)
{
    return(SafeQueue.size());
}

template <class T> unsigned int lqueue<T>::empty(void)
{
    return(SafeQueue.empty());
}

template <class T> void lqueue<T>::erase(void)
{
    lock();
    SafeQueue.erase(SafeQueue.begin(), SafeQueue.end());
    unlock();
}

template <class T> void lqueue<T>::wait(void)
{
    waitEvent();
}

template <class T> void lqueue<T>::broadCast(void)
{
    postEvent();
}
```

参考文献

- Abel, Peter. 1991. *IBM PC Assembly Language and Programming*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall.
- Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1986.
- Andleigh, Prabhat K., and Michael R. Gretzinger. 1992. *Distributed Object-Oriented Data System Design*. Englewood Cliffs, NJ: Prentice Hall.
- Andrews, Mark. 1994. *C++ Windows NT Programming*. New York, NY: M&T Books.
- ANSI Committee Document, 1994. Doc No.X3J16/94-0027 W621/NO414.
- Baker, Louis. 1992. *C Mathematical Function Handbook*. New York, NY: McGraw-Hill.
- Baase, Sara. 1988. *Computer Algorithms: Introduction to Design and Analysis*, 2nd ed. Reading, MA: Addison-Wesley.
- Barr, Avron, and Edward A. Feigenbaum. 1982. *The Handbook of Artificial Intelligence*. Vols. I-II. Los Altos, CA: William Kaufman.
- Behforooz, Ali, and Onkar P. Sharma. 1986. *An Introduction to Computer Science: A Structured Problem Solving Approach*. Englewood Cliffs, NJ: Prentice Hall.
- Benedikt, Michael, ed. 1982. *Cyberspace: First Steps*, 4th ed. Cambridge, MA: MIT Press.
- Berry, John. 1988. *The Waite Group's C++ Programming*. Indianapolis, IN: SAMS.
- Blain, Derrel R., Kurt R. Delimon, and William J. English. 1994. *Real-World Programming for OS/2 2.11*, 2nd ed. Indianapolis, IN: SAMS.
- Blum, Adam. 1992. *Neural Networks In C++: An Object-Oriented Framework for Building Connectionist Systems*. New York, NY: John Wiley & Sons.
- Bolon, Craig. 1986. *Mastering C*. Alameda, CA: Sybex, Inc.
- Booch, Grady. 1994. *Object-Oriented Analysis and Design with Applications*, 2nd ed. Redwood City, CA: Benjamin/Cummings.
- Borger, Egon. 1988. *Trends in Theoretical Computer Science*. Rockville, MD: Computer Science Press.
- Bruce, Phillip, and Sam M. Pederson. 1982. *The Software Development Project Planning and Management*. New York, NY: John Wiley & Sons.
- Budd, Timothy A. 1994. *Classic Data Structures in C++*. Reading, MA: Addison-Wesley.
- Campbell, Joe. 1987. *C Programmer's Guide to Serial Communications*. Indianapolis, IN: SAMS.
- Carroll, Martin D., and Margaret A. Ellis. 1995. *Designing and Coding Reusable C++*. Reading, MA:

- Addison-Wesley.
- Chaitin, G.J. 1987. *Algorithmic Information Theory*. New York, NY: Cambridge University Press.
- Chandy, Mani K, and Stephen Taylor. 1992. *Parallel Programming*. Pasadena, CA: Jones and Bartlett Publishers.
- Clocksin, W.F, and C.S Mellish. 1981. *Programming in Prolog*, 3rd ed. Heidelberg, Berlin: Springer Verlag.
- Conger, James L. 1992. *Windows API Bible: The Definitive Programmer's Reference*. Corte Madera, CA: Waite Group.
- Conger, Jim. 1993. *Windows New Testament*. Corte Madera, CA: Waite Group.
- Conger, Jim. 1993. *Windows API New Testament*. Corte Madera, CA: Waite Group.
- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivset. 1990. *Introduction to Algorithms*. Cambridge, MA: McGraw-Hill/MIT Press.
- Covington, Michael A., Donald Nute, and Andre Vellino. *Prolog Programming in Depth*. Glenview, IL: Scott, Foresman.
- Cox, Brad J. 1986. *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley.
- Custer, Helen. 1993. *Inside Windows NT*. Richmond, WA: Microsoft Press.
- Davis, Phillip J., and Reuben Hersh. 1981. *The Mathematical Experience*. Boston, MA: Houghton Mifflin.
- Davis, William S. 1987. *Operating Systems: A Systematic View*, 3rd ed. Reading, MA: Addison-Wesley.
- Deitel, H.M. 1990. *Operating Systems*. Reading, Mass: Addison-Wesley.
- Dilascia, Paul. 1992. *Windows ++: Writing Reusable Code in C++*. Reading, MA: Addison-Wesley.
- Dorfman, Len. 1990. *Building C Libraries*. Blue Ridge Summit, PA: Windcrest Books.
- Duncan, Ray, Charles Petzold, Andrew Schulman, M. Steven Baker, Ross P. Nelson, Stephen R. Davis, and Robert Moote. 1992. *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, 2nd ed. Reading, MA: Addison-Wesley.
- Eckel, Bruce. 1993. *C++ Inside and Out*. Berkeley, CA: Osborne McGraw-Hill.
- Ellis, Margaret A., and Bjarne Stroustrup. 1990. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley.
- Ellzey, Roy S. 1987. *Computer System Software: The Programmer/Machine Interface*. Chicago, IL: Science Research Associates.
- Ellzey, Roy S. 1982. *Data Structures for Computer Information Systems*. Chicago, IL: Science Research Associates.
- Elson, Mark. 1975. *Data Structures*. Chicago, IL: Science Research Associates.
- Emerson, Frances B. 1987. *Technical Writing*. Boston, MA: Houghton Mifflin.
- Englemore, Robert, and Tony Morgan. 1988. *Blackboard Systems*. Workingham, England: Addison-Wesley.
- Ezzel, Ben. 1990. *Graphics Programming in C++: An Object-Oriented Approach*. Reading, MA: Addison-Wesley.
- Feibel, Werner. 1995. *Complete Encyclopedia of Networking*. Alameda, CA: Novell Press.
- Finkbeiner, Daniel T., and Wendell D. Lindstrom. 1987. *A Primer of Discrete Mathematics*. New York, NY: W. H. Freeman and Company.
- Fischler, Martin A., and Oscar Firschein. 1987. *Intelligence, the Eye, the Brain, and the Computer*. Reading, MA: Addison-Wesley.
- Gibbons, Alan. 1987. *Algorithmic Graph Theory*. New York, NY: Cambridge University Press.
- Glass, Graham. 1993. *UNIX for Programmers and Users: A Complete Guide*. Englewood Cliffs, NJ: Prentice Hall.

- Gordon, Geoffrey. 1969. *System Simulation*. Englewood Cliffs, NJ: Prentice Hall.
- Gorlen, Keith E., Sanford M. Orlow, and Perry Plexico. 1990. *Data Abstraction and Object-Oriented Programming in C++*. New York, NY: John Wiley & Sons.
- Harbison, Samuel P., and Guy L. Steele, Jr. 1991. *C Reference Manual*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall.
- Harmon, Paul, and David King. 1985. *Expert Systems*. New York, NY: John Wiley & Sons.
- Hashim, Safaa A. 1990. *Exploring Hypertext Programming: Writing Knowledge Representation and Problem Solving Programs*. Blue Ridge Summit, PA: Windcrest Books.
- Haviland, Keith, and Ben Salama. 1987. *UNIX System Programming*. Workingham, England: Addison-Wesley.
- Heimlich, Richard, David M. Golden, Ivan Luk, and Peter M. Ridge. 1993. *Sound Blaster: The Official Book*. Berkeley, CA: Osborne McGraw-Hill.
- Holub, Allen I. 1990. *Compiler Design in C*. Englewood Cliffs, NJ: Prentice Hall.
- Hughes, Cameron and Tracey Hughes. 1996. *Collection and Container Classes in C++*. New York, NY: John Wiley & Sons.
- Hughes, Cameron, Thomas Hamilton, and Tracey Hughes. 1995. *Object-Oriented I/O Using C++ Iostreams*. New York, NY: John Wiley & Sons.
- IBM. 1989. *Object-Oriented Interface Design: IBM Common User Access Guidelines*. Carmel, IN: Que.
- IBM. 1992. *OS/2 2.0 Presentation Manager Graphics Programming Guide*. Carmel, IN: Que.
- IBM. 1992. *OS/2 2.0 Control Program Programming Guide*. Carmel, IN: Que.
- Institute of Electrical and Electronic Engineers, Inc. *Information Technology — Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language] IEEE Std. 1003.1*, 1996 Edition ISO/IEC 9945-1. New York, NY: Institute of Electrical and Electronic Engineers, Inc. "POSIX Thread Management Specifications" is reprinted from this publication. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner. Information is reprinted with the permission of the IEEE.
- Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunar Overgaard. 1992. *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Workingham, England: Addison-Wesley.
- Johnson, Eric F., and Kevin Richard. 1994. *Advanced X Windows Applications Programming*, 2nd ed. New York, NY: M&T Books.
- Johnson, Eric F., and Kevin Richard. 1989. *X Windows Applications Programming*. Portland, OR: MIS Press.
- Jones, Capers. 1986. *Programming Productivity*. New York, NY: McGraw-Hill.
- Kaner, Cem. 1988. *Testing Computer Software*. Blue Ridge Summit, PA: Tab Books.
- Kernighan, Brian W., and Dennis M. Ritchie. 1978. *Programming in C*. Englewood Cliffs, NJ: Prentice Hall.
- King, Adrian. 1994. *Inside Windows 95*. Redmond, WA: Microsoft Press.
- Kleiman, Steve, Devang Shah, and Bart Smaalders. 1996. *Programming with Threads*. Mountain View, CA: Sunsoft Press.
- Koeing, Andrew, and Barbara Moo. 1996. *Ruminations on C++*. Menlo Park, CA: Addison-Wesley.
- Kolatis, Maria Shopay. 1985. *Mathematics for Data Processing and Computing*. Reading, MA: Addison-Wesley.
- Korsh, James F., and Leonard J. Garret. 1988. *Data Structures, Algorithms, and Program Style Using C*. Boston, MA: PWS-Kent.
- Kowalski, Robert. 1979. *Logic for Problem Solving*. New York, NY: Elsevier Science.
- Krol, Ed. 1992. *The Whole Internet*, 2nd ed. Sebastopol, CA: O'Reilly & Associates, Inc.
- Kuhn, Robert H., and David A. Padua. 1981. *Tutorial on Parallel Processing*. Silver Spring, MD: IEEE

- Society Press.
- Lai, Robert S., and The Waite Group. 1992. *Writing MS-DOS Device Drivers*, 2nd ed. Reading, MA: Addison-Wesley.
- LaQuey, Tracy, and Jeanne C. Ryer. 1993. *The Internet Companion: Beginner's Guide to Global Networking*. Reading, MA: Addison-Wesley.
- Lewis, Ted. 1995. *Object-Oriented Application Frameworks*. Greenwich, CT: Manning Publications.
- Liebowitz, Jay, and Daniel A. De Salvo. 1989. *Structuring Expert Systems Domain, Design, and Development*. Englewood Cliffs, NJ: Prentice Hall.
- Lorin, Harold. 1972. *Parallelism in Hardware and Software: Real and Apparent Concurrency*. Englewood Cliffs, NJ: Prentice Hall Inc.
- Luse, Marv. 1993. *Bitmapped Graphics Programming in C++*. Reading, MA: Addison-Wesley.
- Mandrioli, Dino, and Carlo Ghezzi. 1987. *Theoretical Foundations of Computer Science*. New York, NY: John Wiley & Sons.
- Matsumoto, Yoshihiro, and Yutaka Ohno. 1989. *Japanese Perspective in Software Engineering*. Singapore: Addison-Wesley.
- Mertin, James, and Kathleen Kavanagh Chapman. 1984. *Local Area Networks Architectures and Implementations*. Englewood Cliffs, NJ: Prentice Hall.
- Meyer, Bertrand. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall.
- Mowbray, Thomas J., and Ron Zahavi. 1995. *The Essential Cobra: Systems Integration Using Distributed Objects*. New York, NY: Wiley & Sons Inc.
- Murray, William D. 1990. *Computer and Digital System Architecture*. Englewood Cliffs, NJ: Prentice Hall.
- Nance, Barry. 1990. *Network Programming in C*. Carmel, IN: Que.
- Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. 1996. *Pthreads Programming*. Sebastopol, CA: O'Reilly & Associates, Inc.
- Orfali, Robert, and Dan Harkey. 1993. *Client/Server Programming with OS/2 2.1*, 3rd ed. New York, NY: Van Nostrand Reinhold.
- Pagan, Frank G. 1991. *Partial Computation and the Construction of Language Processors*. Englewood Cliffs, NJ: Prentice Hall.
- Panov, Kathleen, Larry Salomon, and Arthur Panov. 1993. *The Art of OS/2 2.1 C Programming*. Wellesley, MA: QED.
- Patterson, David A., and John L. Hennessy. 1994. *Computer Organization and Design of the Hardware/Software Interface*. San Mateo, CA: Morgan Kaufmann.
- Peng, Yun, and James A. Reggia. 1990. *Abductive Inference Models for Diagnostic Problem Solving*. New York, NY: Springer Verlag.
- Petzold, Charles. 1992. *Programming Windows 3.1*, 3rd ed. Redmond, WA: Microsoft Press.
- Petzold, Charles. 1994. *OS/2 Presentation Manager Programming*. Emeryville, CA: Ziff Davis Press.
- Pietrik, Matt. 1993. *Windows Internals: The Implementation of the Windows Operating Environment*. Reading, MA: Addison-Wesley.
- Plauger, P.J., and Jim Brodie. 1992. *ANSI and ISO Standard C Programmer's Reference*. Redmond, WA: Microsoft Press.
- Plauger, P.J. 1992. *The Standard C Library*. Englewood Cliffs, NJ: Prentice Hall.
- Pohl, Ira. 1993. *Object-Oriented Programming Using C++*. Redwood City, CA: Benjamin Cummings Publishing Inc.
- Powell, Joel. 1993. *Multitask Windows NT*. Corte Madera, CA: Waite Group.
- Purdin, Jack. 1992. *C Programmer's Toolkit*, 2nd ed. Carmel, IN: Que.
- Reynolds, John C. 1981. *The Craft of Programming*. London, England: Prentice Hall.

- Richter, Jeffrey. 1996. *Advanced Windows: Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95*. Redmond, WA: Microsoft Press.
- Rimmer, Steve. 1992. *Supercharged Bitmapped Graphics*. Blue Ridge Summit, PA: Windcrest/McGraw-Hill Books.
- Robbins, Kay A., and Steven Robbins. 1996. *Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading*. Upper Saddle River, NJ: Prentice Hall.
- Rodrigue, Garry. 1992. *Parallel Computations*. New York, NY: Academic Press.
- Rogers, Jean B. 1986. *A Prolog Primer*. Reading, MA: Addison-Wesley.
- Rubin, Tony. 1988. *User Interface Design for Computer Systems*. New York, NY: John Wiley & Sons.
- Schildt, Herbert. 1987. *Artificial Intelligence Using C*. Berkeley, CA: Osborne McGraw-Hill.
- Schildt, Herbert, and Robert Goosey. 1993. *OS/2 2.0 Programming*. Berkeley, CA: Osborne McGraw-Hill.
- Schulmeyer, Gordon G., and James McManus. 1987. *Handbook—Software Quality Assurance*. New York, NY: Van Nostrand Reinhold.
- Sedgewick, Robert. 1983. *Algorithms*. Reading, MA: Addison-Wesley.
- Seyer, Martin D. 1991. *RS-232 Made Easy: Connecting Computers, Printers, Terminals, and Modems*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall.
- Smith, James T. 1991. *C++ for Scientists and Engineers*. New York, NY: McGraw-Hill.
- Soukup, Jiri. 1994. *Taming C++ Pattern Classes and Persistence for Large Projects*. Reading, MA: Addison-Wesley.
- Staugaard, Andrew C. Jr. 1994. *Structuring Techniques: An Introduction Using C++*. Englewood Cliffs, NJ: Prentice Hall.
- Sterling, Leon, and Ehud Shapiro. 1986. *The Art of Prolog*. Cambridge, MA: MIT Press.
- Stitt, Martin. 1995. *Building Custom Software Tools and Libraries*. New York, NY: John Wiley & Sons.
- Stroustrup, Bjarne. 1991. *The C++ Programming Language*, 2nd ed. Reading, MA: Addison-Wesley.
- Stroustrup, Bjarne. 1994. *The Design and Evolution of C++*. Reading, MA: Addison-Wesley.
- Taligent Press. 1995. *The Power of Frameworks*. Cupertino, CA: Addison-Wesley.
- Teale, Steve. 1993. *C++ IOSTREAMS Handbook*. Reading, MA: Addison-Wesley.
- Tenenbaum, Aaron M., Yedidiah Langsam, and Moshe J. Augenstein. 1992. *Data Structures Using C*. Englewood Cliffs, NJ: Prentice Hall.
- Tenenbaum, Andrew S. 1987. *Operating Systems Design and Implementation*. Englewood Cliffs, NJ: Prentice Hall.
- Thompson, William J. 1992. *Computing for Scientists and Engineers: A Workbook of Analysis, Numerics, and Applications*. New York, NY: John Wiley & Sons.
- Watt, Alan. 1989. *Fundamentals of Three-Dimensional Computer Graphics*. Workingham, England: Addison-Wesley.
- Winston, Patrick Henry, Berthold Klaus, and Paul Horn. *LISP*. Reading, MA: Addison-Wesley.

索引

ADT (abstract data type, 抽象数据类型)	4	规则 (rule)	2, 5
ANSI/ISO 标准	3	过程编程技术 (procedural programming technique)	2
C++进程间通信组件 (IPC)	2	缓冲器 (buffer)	21
iostream 类库	21	基本类 (basic class)	4
streambuf 类	21	基础结构 (infrastructure)	2
标准模板库 (Standard Template Library, STL)	14	集合和容器 (collection and container)	3
并发 (concurrency)	1	集合和容器类 (Collection and Container Class)	11
并行 (parallelism)	1	集合类 (collection class)	4
参数化编程 (parameterized programming)	17	继承 (inheritance)	3
参数化类型 (parameterized type)	3	接口 (interface)	3
抽象类 (Abstract Class)	11	接口类 (interface class)	2
处理器 (processor)	1	接口类 (Interface Class)	11
断言 (assertion)	2, 5	节点类 (Node Class)	11
对象访问策略 (object access policy)	2	结构 (structure)	5
多任务处理 (multitasking)	1	进程 (process)	2
多态 (polymorphism)	3	具体类 (Concrete Class)	11
多线程处理 (multithreading)	1	开发方法学 (methodology)	2
多线程化 (multithreaded)	1	类 (class)	2, 3, 4
多项式 (polynomial)	20	类 calculator, 8	
封装 (encapsulation)	2, 3	类层次 (class hierarchy)	2
格式化 (formatting)	21	类库 (class library)	2, 3, 4
公有 (Public)	13	流 (stream)	21
构造函数 (constructor)	5		

面向对象管道 (object-oriented pipe)	22	动态优先权	35
面向对象架构 (object-oriented architecture)		独占 (monopolize)	30
	2	堆栈 (stack)	35
模式 (pattern)	2	堆栈片断 (stack segment)	25
内联函数 (inline function)	15	对话 (session)	43
内置数值类型	5	对象句柄 (object handle)	25
容器类 (container class)	4	分离进程 (detached process)	41
容器适配器 (container adaptor)	14	分派 (dispatch)	30
软件模型	8	父进程	35
软拼装体 (soft-legoware)	3	挂起-就绪状态 (suspend-ready state)	29
设计思想 (philosophy)	2	挂起-阻塞状态 (suspend-blocked state)	29
实例 (instance)	18	管道 (pipe)	25
受保护 (protected)	13	互斥量 (mutex)	25, 44
受保护接口 (protected interface)	13	环境变量 (environment variable)	26
数据竞争 (data race)	2	唤醒 (wakeup)	30
私有 (Private)	13	恢复 (resume)	32
死锁 (deadlock)	2	激活状态 (active state)	29
位图类 (bitmap class)	7	寄存器 (register)	35
无限延迟 (indefinite postponement)	2	僵化状态 (zombified state)	29
析构函数 (destructor)	5	进程 ID	25
线程 (thread)	1	进程控制块 (process control block)	25
应用框架 (application framework)	2, 4	进程信息块 (process information block)	25
应用类 (Utility Class)	11	进程状态 (process state)	28
有理数 (rational number)	5	静态优先权	35
域类 (Domain Class)	11	就绪状态 (ready state)	28
域模型 (domain model)	2	局部变量 (local variable)	26
运算符重载 (operator overloading)	5	可执行映像 (executable image)	35
增量多线程编程 (incremental multithreading)		空闲 (idle)	29
	2	临界区 (critical section)	44
支持类 (Support Class)	11	轮询 (round-robin)	34
自顶向下 (top-down)	3	上下文切换 (context switch)	35
CPU 时间	26	设备 (device)	25
I/O 计数器	26	时间耗尽 (timerrunout)	30
备用状态 (standby state)	29	时间片 (quantum)	30
程序计数器 (program counter)	35	数据 (data)	25
等待 (waiting)	34	同步执行 (synchronous execution)	39
动态链接库 (dynamic link library)	44	退出码 (exit code)	39

完成 (done)	29	竞争域 (contention scope)	71
文本 (text)	25	可规划单元 (schedulable unit)	70
文件 (file)	25	可执行可规划单元 (schedulable unit of execution)	71
新建 (new)	29	临界区 (critical section)	61
信号量 (semaphore)	44	媒体控制接口 (media control interface)	49
信号量 (semaphores)	25	内存信息转储 (core dump)	60
虚拟控制台 (virtual console)	43	前台进程 (foreground process)	66
硬件资源 (hardware resource)	43	轻量级进程 (lightweight process)	47
优先级 (priority level)	34	守护程序 (daemon)	60
优先类 (priority class)	34	守护进程 (daemon thread)	60
优先权方案 (priority scheme)	34	提升 (raise)	67
运行状态 (running state)	29	同位体 (peer)	48
终止 (terminated)	29	推进 (boost)	67
资源 (resource)	25, 42	危险地带 (danger zone)	60
子-父关系	36	危险区 (danger area)	60
子进程	35	危险页 (danger page)	60
阻塞 (block)	30	先占工作策略 (anticipating-work strategy)	55
阻塞状态 (blocked state)	29	线程 (thread)	47
COM 端口 (COM port)	54	线程池 (thread pool)	72
FIFO (先进先出)	64	线程范例 (thread paradigm)	54
MCI 符号	50	线程块 (thread block)	52
MCI 命令	49, 50	线程信息块 (thread information block)	56
MCI 命令符号	49	协议 (protocol)	67
MIDI 音序器设备	49	休眠 (sleep)	54
Win32 环境	56	延迟工作 (deferring work)	54
单步 (one-shot)	54	一对一 (one-to-one)	71
地址空间 (address space)	52	一进程一线程 (one thread per process)	71
多对一 (many-to-one)	71	永不等待规则 (never-wait rule)	55
分派 (dispatch)	70	用户级线程 (user-level thread)	71
共享资源	52	优先级 (priority level)	58
核心级线程 (kernel-level thread)	71	优先类 (priority class)	58
后台进程 (background process)	66	优先权 (priority)	58
基优先权 (base priority)	66	远程线程 (remote thread)	60
监视线程 (monitoring thread)	54	运行时系统 (runtime system)	71
解析器 (parser)	49	先占工作 (anticipating work)	54
进程信息块 (process information block)	52	执行线程 (thread of execution)	49
警戒页 (guard page)	60		

主线程 (main thread 或 primary thread)	49	unidirectional dependency)	90
自然语言处理 (natural language processing, NLP)	49	地址空间 (address space)	114
FIFO	86	动态数据交换 (dynamic data exchange)	109
处理器池 (pool of processor)	81	动态数据交换对话 (dynamic data exchange session, DDE)	109
低级别规划 (low-level scheduling)	84	队列 (queue)	117
多程序编程 (multiprogramming)	73	多重集 (multiset)	117
多处理器处理 (multiprocessing)	80	多重映射 (multimap)	117
多任务 (multitask)	73	管道 (pipe)	98
多任务编程 (multitasking)	73	集 (set)	117
多线程 (multithread)	73	进程间或线程间通信 (interprocess or interthread communication)	94
阈值 (threshold value)	88	进程间通信机制 (interprocess communication mechanism)	114
合作 (cooperation)	75	空关系 (null relationship)	91
紧密耦合 (tightly coupled)	81	链表 (linked list)	117
进程级多任务编程 (process-level multitasking)	75	命令行参数 (command-line argument)	96
轮询 (round-robin)	86	命名管道 (named pipe)	98
抢占 (preemption)	75	匿名管道 (anonymous pipe)	98
上下文切换 (context switch)	75	全局变量 (global variable)	114
时间片 (quantum)	79	矢量 (vector)	117
时间片断 (slice)	78	树 (tree)	117
实时 (real-time)	85	双端队列 (deque)	117
同步 (synchronization)	82	双向依赖性 (bidirectional dependency)	90
同时 (simultaneous)	82	通信依赖性 (communication dependency)	90
协处理器 (coprocessor)	81	同步化 (synchronized)	119
用户控制多任务 (user-controlled multitask)	74	同时 (simultaneous)	119
主-次线程模型 (primary-secondary thread model)	83	图表 (graph)	117
最短任务优先规划法 (shortest-job-first scheduling)	87	线程间通信 (interthread communication)	114
最短剩余时间规划法 (shortest-remaining-time)	88	依赖关系 (dependency relationship)	89
IPC (interprocess communication, 进程间通信)	95	依赖性矩阵 (dependency matrix)	93
参数传递 (parameter passing)	119	映射 (map)	117
单向依赖性 (one-way dependency 或		主题-项对 (topic-item pair)	114
		FF (finish-to-finish)	129
		FS (finish-to-start)	129
		SF (start-to-finish)	129
		SS (start-start)	129

不可分割性 (indivisibility)	132	包容关系 (containment relationship)	176
定时信号量 (timed semaphore)	148	标准模板库 (Standard Template Library)	152
二进制信号量 (binary semaphore)	131	操作系统 API	149
广播机制 (broadcast mechanism)	141	成员函数 (member function)	149
后置条件 (postcondition)	128	纯虚拟 (pure virtual)	155
互斥 (mutual exclusion)	133	打开模式 (open mode)	175
互斥等待信号量 (mutex wait semaphore)	132	代码和数据聚集 (code and data aggregation)	150
互斥信号量 (mutex semaphore)	132	单程序多数数据流 (SPMD)	157
竞争条件 (race condition)	124	单指令多数数据流 (SIMD)	157
临界区 (critical section)	130	独立域接口 (domain-independent interface)	151
临界区数据类型 (Critical Section data type)	145	对象 (object)	162
前置条件 (precondition)	128	多对多 (many-to-many)	175
取消锁定 (unlock)	140	多对一 (many-to-one)	175
任务同步 (Task Synchronization)	126	多指令单数据流 (MISD)	157
设备同步 (Device Synchronization)	126	多指令多数数据流 (MIMD)	157
事件互斥量 (event mutex)	141, 143	翻译组件 (translation component)	162
事件信号量 (event semaphore)	132	访问权限 (access right)	174
释放 (free)	140	服务器进程 (server process)	174
释放 (release)	140	复合 (composition)	175
数据竞争 (data race)	124	管道模式 (pipe mode)	175
数据同步 (Data Synchronization)	126	过程库 (procedure library)	149
死锁 (deadlock)	124	缓冲器组件 (buffer component)	161
条件变量 (condition variable)	141, 143	监听模式 (listening mode)	179
条件变量 (conditional variable)	130	接口类 (interface class)	149
无限延迟 (indefinite postponement)	145	具体类 (concrete class)	150
信号量 (semaphore)	130	客户 (机) 进程 (client process)	174
原子性 (atomicity)	132	蓝图接口类 (blueprint interface class)	154
滞后时间 (lag time)	130	类类型 (type of class)	149
周边设备 (peripheral device)	123	面向对象流 (complete object-oriented stream)	162
Container (容器)	154	命名管道 (named pipe)	158, 174
filebuf 类	163	目标 (destination)	162
fstream 对象	161	内联 (inline)	154
iostream 层次	161	匿名管道 (anonymous pipe)	158, 165
ios 组件	166	排它性读 (exclusive read)	162
pstream 对象	161		
streambuf 类	163		

排它性写 (exclusive write)	162	EREW(排它性读和排它性写)	248
适配器类 (adaptor class)	151	FTP (File Transfer Protocol, 文件传输协议)	
数据缓冲器 (data buffer)	160		225
私有 (private)	177	LIPS (logical inferences per second, 每秒逻辑推理数)	232
虚函数 (virtual function)	150	PRAM (并行随机存取机器, Parallel Random Access Machine)	248
一对多 (one-to-many)	175	VTOC (内容可视表)	238
一对一 (one-to-one)	175	并行算法 (parallel algorithm)	222
源 (source)	162	常量成员函数 (const member function)	251
重用 (reuse)	151	地址端口 (address port)	246
状态组件 (state component)	162	动态数据交换 (dynamic data exchange, DDE)	
追加 (append)	162		224
字节流 (stream of bytes)	162	对象访问策略 (object-access policy)	245
FF (finish-to-finish)	205	多线程软件架构 (multithreaded software architecture)	221
FIFO (先进先出)	198	分布式专家系统 (distributed expert system)	
持久性 (durability)	216		230
队列 (queue)	198	分解软件 (factoring software)	221
方法 (method)	204	公共资源 (common resource)	234
孤立性 (isolation)	216	过程化编程方式 (procedural style of programming)	238
互斥量持续时间 (mutex duration)	195	黑板 (blackboard)	235
类聚集 (class aggregation)	219	计算机程序 (computer program)	222
命名互斥量 (named mutex)	195	计算机汇编语言 (assembly language)	246
匿名互斥量 (anonymous mutex)	195	寄存器 (register)	246
设备类 (device class)	219	架构 (architecture)	221
事件互斥量 (event mutex)	213	离散事件模型 (discrete event model)	246
事件系统服务 (event system service)	210	临界区 (critical section)	234
事务 (transaction)	216	逻辑子过程 (logical subprocedure)	238
数据 (data class)	219	面向对象客户机/服务器 (object-oriented client/server)	246
双端队列 (deque)	198	设备上上下文 (device context)	247
同步类 (synchronization class)	219	事件处理器 (event handler)	234
析构 (destruction)	186	事务服务器 (transaction server)	229
一致性 (consistency)	216	数据库管理系统 (database management system)	246
异常处理 (exception handling)	188		
原子性 (atomicity)	216		
CRCW(并发读和并发写)	248		
CREW(并发读和排它性写)	248		
DMA 通道 (直接存储存取通道)	246		
ERCW(排它性读和并发写)	248		

算法 (algorithm)	222	继承 (inheritance)	255
图解搜索算法 (graph search algorithm)	231	节点类 (node class)	260
图形用户界面 (graphical user interface, GUI)	228	结构 (struct)	261
文件服务器 (file server)	225	类层次 (class hierarchy)	255
物理设备驱动程序 (physical device driver)	247	类库 (class library)	277
消息模型 (message model)	234	列表类 (List Class)	261
虚拟设备驱动程序 (virtual device driver)	247	面向对象数据库管理系统 (OODBS, object-oriented database management system)	269
哑终端 (dumb terminal)	229	签名分辨率 (signature resolution)	260
语义学 (semantics)	236	强制转换基本元素 (casting primitive)	269
语用学 (pragmatics)	236	容器类 (container class)	261
增量多线程处理 (incremental multithreading)	253	入口点 (entry point)	272
增量途径 (incremental approach)	252	深层复制构造函数 (deep copy constructor)	260
逐步求精法 (stepwise refinement approach)	238	数组和矢量类 (Array and Vector Class)	261
主机-远程端 (host-remote) 模式	229	双端队列类 (Deque Class)	261
主线程 (main thread)	234	双向通信 (two-way communication)	270
子指令 (subinstruction)	222	宿主类 (host class)	269
自顶向下 (top-down)	238	线程句柄 (thread handle)	272
FF (finish-to-finish)	275	线程类 (thread class)	269
ITC (线程间通信, interthread communication)	261	相关联动动作序列 (sequence of interrelated actions)	277
STL (Standard Template Library, 标准模板库)	261	异常处理 (exception handling)	260
this 指针	269	引用 (reference)	260
并集 (set union)	274	友元成员函数 (friend member function)	269
抽象基类 (abstract base class)	255	域对象 (domain object)	269
队列和优先权队列类 (Queue and Priority Queue Class)	261	域类 (domain class)	269
多线程服务器 (minimultithreaded server)	268	运行时多态 (runtime polymorphism)	260
多线程集服务器 (multithreaded set server)	268	指针 (pointer)	260
过程库 (procedure library)	277	属性 (attribute)	272
互斥和事件类 (mutex and event class)	269	SS (start-to-start)	307
		保护 (protection)	309
		迭代器 (iterator)	320
		堆 (heap)	296
		堆栈片断 (stack segment)	296
		翻译单元 (translation unit)	296

封装 (encapsulation)	309	任务准则优先权 (mission-critical priority)	341
函数作用域 (function scope)	295	软件测试 (software test)	323
结构化线程取消 (structured thread cancellation)	310	软件故障 (software failure)	323
局部作用域 (local scope)	295	软件确认 (software validation)	326
类作用域 (class scope)	295	软件验证 (software verification)	326
连接 (linkage)	296	深层复制 (deep copy)	337
内部连接 (internal linkage)	296	实时优先权 (real-time priority)	341
容错 (fault tolerance)	311	说明性编程技术 (declarative programming technique)	341
容器类 (container class)	320	同事模型 (work crew model)	342
算法 (algorithm)	320	谓词演算 (predicate calculus)	341
通信依赖性 (communication dependency)	298	应用框架 (application framework)	334
同步依赖性 (synchronization dependency)	298	主/次线程模型 (primary/secondary thread model)	342
外部连接 (external linkage)	296	资源过剩 (resource surplus)	331
文件作用域 (file scope)	295	组合理论 (combinatorial theory)	341
无限延迟 (indefinite postponement)	306	FIFO 规划策略	350
线程安全 (thread safe)	318	SCHED_OTHER 策略	351
异常处理器 (exception handler)	315	超时运行 (overrun)	369
应用框架 (application framework)	299	规划参数 (scheduling parameter)	349
重复进入代码 (reentrant code)	319	量化误差 (quantization error)	369
背景优先权 (background priority)	341	取消点 (cancellation point)	381
操作上的测试 (operational test)	323	取消控制接口 (cancellation control interface)	380
测试计划 (test plan)	328	取消能力类型 (cancelability type)	381
测试实例 (test case)	328	取消能力状态 (cancelability state)	381
单元测试 (unit testing)	324	取消清理处理器 (cancellation cleanup handler)	382
对等模型 (peer model)	342	实时信号扩展 (Realtime Signals Extension)	365
工作管道模型 (work pile model)	342	实时信号扩展 (Realtime Signals Extension)	369
规范测试 (specification testing)	335	信号号 (signal number)	365
回归测试 (regression test)	333	延迟取消能力集 (deferred cancelability set)	381
集成测试 (integration testing)	324	语言无关线程取消功能 (language-independent thread cancellation functionality)	385
拷贝构造函数 (copy constructor)	337		
可接受测试 (acceptance testing)	335		
逻辑语义 (logic semantics)	341		
模式类 (pattern class)	334		
普通优先权 (normal priority)	341		
浅层复制 (shallow copy)	337		
强度测试 (stress testing)	328		